Drie valkuilen bij de prestatie-evaluatie van Javaprogramma's

Three Pitfalls in Java Performance Evaluation

Andy Georges

Promotoren: prof. dr. ir. K. De Bosschere, prof. dr. ir. L. Eeckhout
Proefschrift ingediend tot het behalen van de graad van
Doctor in de Ingenieurswetenschappen: Computerwetenschappen

Vakgroep Elektronica en Informatiesystemen
Voorzitter: prof. dr. ir. J. Van Campenhout
Faculteit Ingenieurswetenschappen
Academiejaar 2007 - 2008

UNIVERSITEIT
GENT

ISBN 978-90-8578-197-4
NUR 980
Wettelijk depot: D/2008/10.500/16

For Veerle, always the light in my life,
For Elias, firstborn, who gave profound meaning to existence
For Nathan, who showed that there's no such thing
as too much of something good,
For I would not be complete without you.

Acknowledgements

At the onset of this work, the road was unclear. The general direction was known, but there was no telling where we would end up. If it was, it would not be called research. To walk this path alone is not possible, or to quote Newton '*If I have been able to see further than others, it is because I have stood on the shoulders of giants*'. I am much indebted to my promotors, professor Koen De Bosschere and professor Lieven Eeckhout, for offering me the chance to start working at the ELIS department, and to encourage me to start this research. Thank you for your councel, help, understanding that not all things tried eventually work out, and to keep motivating me to run the race until the finish.

Every doctoral dissertation has need of a jury (in no particular order): Prof. Koen De Bosschere, Prof. Lieven Eeckhout, Prof. Kathryn McKinley, Prof. Matthias Hauswirth, Prof. Jan Van Campenhout, Prof. Kris Coolsaet, Prof. Yolande Berbers, and Prof. Verhoeven. Thank you, your comments and observations definitely have changed this thesis for the better.

I would also like to express my gratitude to the people with whom I have worked. It was cool working with you. In particular, I would like to thank Dries Buytaert for the great times we had working on several papers – at times even into the dark hours of night – and for introducing me to a cool bunch of Drupaling people. A lot of people have contributed to the knowledge I have assembled these years, among which are: Dries Buytaert, Michiel Ronsse, Jonas Maebe, Mark Christiaens. I will try not to forget. I shared good times with the guys crowding our office: Kris Venstermans and Davy Genbrugge. Thanks!

Writing down this disseration has been quite a learning experience. I would like to express my gratitude to those few who proofread it back to back: Koen De Bosschere, Lieven Eeckhout, Kenneth Hoste and Dries Buytaert. Without you this work would not have become what it is today.

I have been working on several projects the past years, and I am indebted to The Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT) as well as to Ghent University for supporting me.

Finally, I would like to thank my wife Veerle, my sons Elias and Nathan, my family and my friends for their support, for understanding that deadlines

are there to be met and for sticking with me. You are awesome!

Andy Georges
Ghent, April 30, 2008

Examencommissie

Prof. Yolande Berbers
Departement Computerwetenschappen,
Faculteit Ingenieurswetenschappen
Katholieke Universiteit Leuven

Prof. Kris Coolsaet
Vakgroep TWI, Faculteit Wetenschappen
Universiteit Gent

Prof. Koen De Bosschere, promotor
Vakgroep ELIS, Faculteit Ingenieurswetenschappen
Universiteit Gent

Prof. Lieven Eeckhout, promotor
Vakgroep ELIS, Faculteit Ingenieurswetenschappen
Universiteit Gent

Prof. Matthias Hauswirth
Faculty of Informatics
University of Lugano, Switzerland

Prof. Kathryn McKinley
Department of Computer Sciences
The University of Texas at Austin, USA

Prof. Jan Van Campenhout, secretaris
Vakgroep ELIS, Faculteit Ingenieurswetenschappen
Universiteit Gent

Prof. Ronny Verhoeven, voorzitter
Onderwijsdirecteur Faculteit Ingenieurswetenschappen
Universiteit Gent

Leescommissie

Prof. Yolande Berbers
Departement Computerwetenschappen,
Faculteit Ingenieurswetenschappen
Katholieke Universiteit Leuven

Prof. Kris Coolsaet
Vakgroep TWI, Faculteit Wetenschappen
Universiteit Gent

Prof. Lieven Eeckhout, promotor
Vakgroep ELIS, Faculteit Ingenieurswetenschappen
Universiteit Gent

Prof. Matthias Hauswirth
Faculty of Informatics
University of Lugano, Switzerland

Prof. Kathryn McKinley
Department of Computer Sciences
The University of Texas at Austin, USA

Samenvatting

Gedurende het laatste decenium heeft de programmeertaal Java een immense groei doorgemaakt. Dit werd grotendeels mogelijk gemaakt door de infrastructuur die nodig is om Java-applicaties uit te voeren: de Java virtuele machine (VM). Een dergelijke VM zorgt ervoor dat applicaties, geschreven in Java, kunnen worden overgebracht van het ene naar het andere (fysieke) platform. De reden hiervoor is dat de VM de applicaties afschermt van het onderliggende systeem.

Java-applicaties worden gecompileerd van Java broncode naar een intermediaire voorstelling, bytecode genoemd. Een dergelijke applicatie bestaat dan uit een aantal klassebestanden, die worden ingeladen door de virtuele machine als ze vereist worden om de applicatie uit te voeren. Zodra een methode uit de applicatie wordt opgeroepen, zal de virtuele machine deze ofwel interpreteren, ofwel doorcompileren naar de machinecode van het platform waarop de applicatie wordt uitgevoerd. De VM zorgt verder voor het plannen van de uitvoering van Javadraden, de geheugensanering, het optimaliseren van hete code, enz. Tijdens de uitvoering van een Java-applicatie komt de VM dus geregeld tussenbeide om de interne huishouding te regelen en ervoor te zorgen dat de applicatie zo snel mogelijk wordt uitgevoerd. Bovendien gedraagt een VM zich gewoonlijk op een niet-deterministische wijze.

Hiermee doelen we niet zozeer op de semantiek van de applicatie, maar wel op de interne beslissingen die de VM neemt om die applicatie zo snel en efficiënt mogelijk uit te voeren. Een concreet voorbeeld is de optimalisatie van hete code, wat nodig is om prestatie te halen die competitief is in vergelijking met programma's geschreven in klassieke programmeertalen, zoals C. Om zo weinig mogelijk perturbatie te veroorzaken zal een VM gewoonlijk stalen (*samples*) nemen van de methoden die worden uitgevoerd. Dit wordt op geregelde tijdstippen gedaan, maar door kleine verschillen in de timing is het best mogelijk dat over verschillende uitvoeringen van eenzelfde programma niet steeds dezelfde methoden worden gezien in de stalen, en dat deze dus ook niet (op hetzelfde ogenblik) worden geoptimaliseerd. Hierdoor verschilt de verzameling van geoptimaliseerde methoden van uitvoering tot uitvoering. Andere factoren spelen ook een belangrijke rol, zoals de geheugensanering en het plannen van de draadwissels. Daar waar prestatie-analyse van Java op het eerste gezicht een eenvoudige zaak lijkt, blijkt het bij nader in-

zien toch een vrij complexe aangelegenheid te zijn die nauwkeurige aandacht verdient.

In dit proefschrift spitsen we ons voornamelijk toe op het gedrag en de prestatie van Java-applicaties. Gedurende het werk dat aanleiding gaf tot dit doctoraatsproefschrift, hebben we drie valkuilen ontdekt die door vorige onderzoekers over het hoofd werden gezien toen ze de prestatie van Java-applicaties analyseerden.

De eerste valkuil die aangekaart wordt, houdt verband met de complexe interactie tussen de virtuele machine, de applicatie en de invoer die door de applicatie verwerkt wordt. We tonen aan dat het gedrag voor kort lopende applicaties voornamelijk afhankelijk is van de virtuele machine. Vóór langer lopende applicaties wordt het gedrag eerder bepaald door de applicatie zelf, hoewel de invloed van de virtuele machine merkbaar blijft. Door gebruik te maken van statistische analyse, meer bepaald van principale componenten-analyse en clusteranalyse, tonen we het bestaan van deze valkuil aan. We gebruiken hiervoor een groot aantal prestatiekarakteristieken die gemeten werden met behulp van de prestatietellers van de microprocessor, op vijf virtuele machines en voor veertien benchmarks. Elke applicatie heeft zowel een klein als een groot stel invoergegevens. Klaarblijkelijk kunnen we op basis van de bemeten karakteristieken de werklasten clusteren; voor kort lopende werklasten gebeurt de clustering quasi helemaal volgens de virtuele machine, terwijl bij langer lopende werklasten de applicatie zelf een grotere invloed uitoefent. Zeker bij benchmarks uit de SPECjvm98 suite is het zo dat ook voor langer lopende werklasten, de uitvoering nog steeds sterk wordt beïnvloed door de virtuele machine. Bovendien zijn kortlopende werklasten niet representatief voor langer lopende werklasten en zijn de metingen niet overdraagbaar tussen virtuele machines.

Dit werk heeft aangetoond dat er nood is aan grotere en langer lopende benchmarks. Hieraan wordt deels tegemoet gekomen door de DaCapo suite, die na ons werk gepubliceerd werd. We hebben ook aangetoond dat men nauwgezet moet overwegen hoe het experiment wordt ontworpen, om te vermijden dat men de virtuele machine bemeet in plaats van de applicatie. Voor dit werk gepubliceerd werd, gebruikte men frequent simulaties om het gedrag van Java-applicaties te bestuderen, waardoor men eerder kort lopende werklasten gebruikte om de meettijden binnen de perken te houden.

De tweede valkuil die we onder de loep houden, vinden we terug bij de daadwerkelijke analyse van prestatiemetingen. Na een grondige literatuurstudie van 50 artikels, die gedurende de laatste zeven jaar op topconferenties zoals OOPSLA, PLDI, CGO, ISMM en VEE gepubliceerd werden, bleek dat er een brede waaier aan technieken wordt gebruikt. Dit is het geval zowel bij het opzetten van experimenten – wat de keuze van virtuele machine(s), benchmarks, grootte van het grabbelgeheugen, enz. inhoudt – als bij de analyse van de verkregen data – waar men, bijvoorbeeld, de beste prestatiemeting uit 3 experimenten gebruikt.

Dergelijke prevalente methoden worden vervolgens gebruikt voor het rap-

porteren van prestatie, alsook om nieuwe technieken met bestaand werk te vergelijken en conclusies i.v.m. de verbetering qua prestatie door deze techniek te trekken. Gezien de uitvoering van Java-applicaties gepaard gaat met niet-determinisme in de virtuele machine, hoeft het niet te verbazen dat het gebrek aan statistische rigoureusheid in de verscheidene prevalente methoden aanleiding kan geven tot misleidende of zelfs verkeerde conclusies. Hiermee bedoelen we dat de conclusies niet representatief zijn voor wat in werkelijkheid gebeurt, of dat ze een vertekend beeld geven van de realiteit die het best benaderd wordt door een statistisch model. Om deze valkuil te omzeilen stellen we een rigoureuze statistische aanpak voor. In wezen gebruiken we confidentie-intervallen voor het rapporteren van prestatiemetrieken alsook om vergelijkingen tussen alternatieve technieken te maken, zoals bijvoorbeeld het vergelijken van een nieuw geheugensaneringsalgoritme met bestaande algoritmen. We beweren tevens dat het belangrijk is om voldoende metingen te verrichten om betrouwbare resultaten te verkrijgen.

Er is een manier om het niet-determinisme, dat veroorzaakt wordt door de tijdsgebaseerde optimalisatie-infrastructuur in een virtuele machine te elimineren: het herspelen van compilatie – *replay compilation*. Deze techniek wordt als volgt toegepast. Tijdens een eerste profileringsuitvoering van de applicatie houdt men bij welke methoden door de virtuele machine worden geoptimaliseerd in een zogenaamd compilatieplan. Gedurende een tweede uitvoering worden twee iteraties van de applicatie gebruikt: in de eerste iteratie worden alle methoden uit het compilatieplan geoptimaliseerd, in de tweede iteratie wordt het optimalisatiemechanisme uitgeschakeld, en zal men de eigenlijke meting verrichten. De huidige trend om zich te beperken tot een enkel compilatieplan blijkt niet representatief voor het daadwerkelijk gebruik van meerdere compilatieplannen tijdens het experimenteren, ook al wordt dat plan samengesteld uit meerdere plannen waarbij de methoden worden behouden die in de meerderheid van de gevallen werden geoptimaliseerd. De reden hiervoor is dat de variabiliteit die we observeren in de plannen zelf te groot is. Het gebruik van meerdere compilatieplannen maakt het mogelijk om een *matched-pair* analyse te gebruiken, wat resulteert in smallere confidentieintervallen voor de gemiddelde waarde van de onderzochte prestatiemetriek.

Tenslotte heeft de derde valkuil betrekking op het gebruik van een globale prestatiemaat om een applicatie of een virtuele machine te analyseren. We tonen aan dat Java-applicaties fasegedrag vertonen op het niveau van hun methoden. Dit betekent dat de verschillende instanties van een bepaalde methode meer gelijkenissen vertonen met elkaar dan met instanties van andere methoden tijdens de uitvoering, wat hun gedrag op micro-architecturaal niveau betreft. Een fase correspondeert dan met een verzameling deelbomen uit de dynamische oproepboom, die alle dezelfde top hebben, nl. de methode die de fase identificeert. We stellen een 2-delig algoritme voor waarmee prestatiemetrieken op hardware niveau gecorreleerd kunnen worden met de methoden uit de applicatie. In de eerste stap worden fasen gedetecteerd, en in de tweede stap gebeurt de eigenlijke meting van verscheidene prestatie-

metrieken voor elk van de fasen. Hiermee kan een programmeur zien welke methoden zich slechter dan gemiddeld gedragen, bijvoorbeeld in het aantal data cache missers die ze oplopen.

In dit proefschrift besteden we veel aandacht aan statistische rigoureusheid. Bij elke valkuil onderbouwen we onze stelling a.d.h.v. statistiek. Hopelijk zal dit werk andere onderzoekers motiveren om hun experimenten ook op een rigoureuze manier op te zetten.

Summary

The Java programming language has known a remarkable growth over the last decade. This is partially due to the infrastructure required to run Java applications on general purpose microprocessors: a Java virtual machine (VM). The VM ensures that Java applications are portable across different hardware platforms, because it shelters the applications from the underlying system. Hence the motto *write once, run (almost) anywhere*.

Java applications are compiled to an intermediate form, called bytecode, and consist of a number of so-called class files. The virtual machine takes care of class loading, interpreting or compiling the bytecode to the native code of the underlying hardware platform, thread scheduling, garbage collection, etc. As such, during the execution of a Java application, the VM regularly intervenes to take care of housekeeping tasks and to optimise the application as it is executing. Furthermore, the specific implementation details of most virtual machines insert non-deterministic behaviour, not into the semantic part of the execution, but rather into the lower level execution. For example, to bring a Java application up to competitive speed with classical compiled programs written in languages such as C, the virtual machine needs to optimise Java bytecode. To limit the execution overhead, most virtual machines use a time sampling mechanism to determine the hot methods in the application. This introduces non-determinism, as over several runs, the methods are not always optimised at the same moment, nor is the set of optimised methods always the same. Other factors that introduce non-determinism are the thread scheduling, garbage collection, etc. It is readily seen that performance analysis of Java applications is not as simple as it seems at first, and warrants closer inspection.

In this dissertation we are mainly interested in the behaviour of Java applications and their performance. In the course of this work, we uncovered three major pitfalls that were not taken into account by researchers when analysing Java performance prior to this work. We will briefly summarise the main achievements presented in this dissertation.

The first pitfall we present involves the interaction between the virtual machine, the application and the input to the application. The performance for short running applications is shown to be mainly determined by the virtual machine. For longer running applications, this influence decreases, but

remains tangible. We use statistical analysis, such as principal components analysis and cluster analysis (K-means and hierarchical clustering) to demonstrate and clarify the pitfall. By means of a large number of performance characteristics measured using hardware performance counters, five virtual machines and fourteen benchmarks with both a small and a large input size, we demonstrate that short running workloads are primarily clustered by virtual machines. Even for long running applications from the SPECjvm98 benchmark suite, the virtual machine still exerts a large influence on the observed behaviour at the microarchitectural level. This work has shown the need for both larger and longer running benchmarks than were available prior to it – this was (partially) met by the introduction of the DaCapo benchmark suite – as well as a careful consideration when setting up an experiment to avoid measuring the virtual machine, rather than the benchmark. Prior to this work, people were quite often using simulation with short running applications (to save time) for exploring Java performance.

The second pitfall we uncover involves the analysis of performance numbers. During a survey of 50 papers published at premier conferences, such as OOPSLA, PLDI, CGO, ISMM and VEE, over the past seven years, we found that a variety of approaches are used, both for experimental design – for example, the input size, virtual machines, heap sizes, etc. – and, even more importantly, for data analysis – for example, using a best out of 3 performance number. New techniques are pitted against existing work using these prevalent approaches, and conclusions regarding their successfulness in beating prior state-of-the-art are based upon them. Given the fact that the execution of Java applications usually involves non-determinism in the virtual machine – for example, when determining which methods to optimise – it should come as no surprise that the lack of statistical rigour in these prevalent approaches leads to misleading or even incorrect conclusions. By this we mean that the conclusions are either not representative of what actually happens, or even contradict reality, as modelled in a statistical manner. To circumvent this pitfall, we propose a rigorous statistical approach that uses confidence intervals to both report and compare performance numbers. We also claim that sufficient experiments should be conducted to get a reliable performance measure.

The non-determinism caused by the timer-based optimisation component in a virtual machine can be eliminated using so-called replay compilation. This technique will record a compilation plan during a first execution or profiling run of the application. During a second execution, the application is iterated twice: once to compile and optimise all methods found in the compilation plan, and a second time to perform the actual measurement. It turns out however that current practice of using either a single plan – corresponding to the best performing profiling run – or a combined plan choosing the methods that were optimised in, say, more than half the profiling runs, is no match for using multiple plans. The variability observed in the plans themselves is too large to capture in one of the current practices. Consequently, using multiple plans is definitely the better option. Moreover, this allows using a matched-

pair approach in the data analysis, which results in tighter confidence intervals for the mean performance number.

The third pitfall we examine is the usage of global performance numbers when tuning either an application or a virtual machine. We show that Java applications exhibit phase behaviour at the method level. This means that instances of the same method show more similarity to each other, behaviour-wise, than to instances of other methods. A phase can then be identified as a set of sub-trees of the dynamic call-tree, with each sub-tree headed by the same method. We present an two-step algorithm that allows correlating hardware performance counter data in step 2 with the phases determined in step 1. The information obtained can be applied to show the programmer which methods perform worse than average, for example with respect to the number of cache misses they incur.

In the dissertation, we pay particular attention to statistical rigour. For each pitfall, we use statistics to demonstrate its presence. Hopefully this work will encourage other researchers to use more rigour in their work as well.

Contents

List of Tables

List of Figures

List of Abbreviations

ANOVA	Analysis Of Variance
API	Application Programming Interface
CA	Cluster Analysis
CopyMS	Copy Mark Sweep
CPI	Cycles Per Instruction
CPU	Central Processing Unit
GC	Garbage Collection
GenCopy	Generational Copying
GenMS	Generational Mark Sweep
HPM	Hardware Performance Monitors
JIT	Just In Time
JVM	Java Virtual Machine
L1 D-cache	Level 1 data cache
L1 I-cache	Level 1 instruction cache
OS	Operating System
PAPI	Performance Application Programming Interface
PCA	Principal Components Analysis
TLB	Translation Lookaside Buffer
RVM	Research Virtual Machine
VEE	Virtual Execution Environment
VM	Virtual Machine

Prologue

And I, even I, turned toward all the works of mine that my hands had done and toward the hard work that I had worked hard to accomplish, and, look! everything was vanity and a striving after wind, and there was nothing of advantage under the sun. — **Ecclesiastes 2:11**

Chapter 1

Introduction

A beginning is the time for taking the most delicate care that the balances are correct.
– **Bene Gesserit proverb**

1.1 Java, both language and platform

In the early 1990s, a secret effort took place in the SUN laboratories[1] that would lead to the existence of the programming language now commonly known as Java. At first, Java – then still disguised under the names Green, Oak and FirstPerson[2] – was targeted at embedded devices or consumer electronics platforms. Officially announced in 1995 at Sun World, Java rapidly grew to prominence together with the advent of the Internet and its associated technologies. One of the original goals set for Java was platform independence. It should thus be possible to run a Java application on any platform. This is achieved by compiling Java source to an intermediate form, known as bytecode. Yet, there is no such thing as a free lunch; consequently, there is a price to be paid to achieve said platform independence. Usually the Java bytecode cannot be natively executed on a general-purpose CPU. Execution of a Java application requires the mediation of a Java virtual machine (JVM) that loads the bytecode and transforms it into a format that can be understood by the CPU on top of which it runs. This is achieved by either interpreting the bytecodes one by one, and thus simulating the Java stack machine, or by compiling the bytecode to native machine code. A bonus of using a virtual machine is the possibility to further optimise the native code at run time. The virtual machine also takes care of garbage collection, thread scheduling, class loading, type checking, etc.

In the years since it was first announced, a lot has been said and written on

[1] `http://www.java.com/en/about/`
[2] `http://www.java.com/en/javahistory/timeline.jsp`

Java[3]. It is claimed that, by using a virtual machine, Java is (too) slow. It could never be used for development of large software applications. People disagree with its object model, with its API, etc. But, despite all the criticisms uttered, the importance and impact of Java kept growing and nowadays most people cannot imagine an information technology world without it. Libraries were developed or expanded, and a lot of tools were built. Today, Java applications are truly omnipresent. A multitude of programs have been written for high-end application servers, web servers, desktop machines, and tiny hand-held devices. Numerous institutions – including our own Alma Mater – educate future computer scientists using Java[4].

Initially the only available virtual machine was provided by SUN, but it did not take long for other virtual machines capable of running Java applications to emerge. Some of these were based on technology licensed from SUN, such as the first IBM virtual machines, or the Blackdown port of SUN's HotSpot 1.x virtual machines to the Linux platform. Others were cleanroom – untainted by SUN virtual machine code or contributions from people who had peeked at the SUN virtual machine code – virtual machines, such as SableVM [91] or Kaffe [63]. It is even possible to write a Java virtual machine in (mostly) Java, as proved by the Jikes RVM project [3]. While the Java application can remain largely ignorant[5] of which virtual machine it runs on, there is possibly a large difference in behaviour and performance depending on the VM.

With the advent of multiple virtual machines, tools, associated libraries, and development environments, Java became more than simply a language. And as the interest in the Java platform grew, a need to understand the applications that thrived on it became apparent. People, who have always been an inquisitive and inventive species, rapidly tried to improve on what they perceived as shortcomings of the Java platform. With SUN opening up its Java implementation, and other languages being ported (e.g., Jython which implements a Python interpreter in Java, allowing easy inter-operation between both languages) to run on the JVM, interest in the platform will probably not fade any time soon. In fact, Gartner [46] predicts that in the near future – by 2010 – 80% of all new software will be written in languages with a managed runtime system (Java or C#).

1.2 Research into Java performance

Researchers became enamoured with the Java concept, as it provided new opportunities from both a language design point and from a virtualisation

[3] Language advocates often have a mindset that is not unlike the mindset of religious zealots.

[4] Whether this will prove to be beneficial is a question that is open for dispute.

[5] It should be noted that, while platform independence is achieved through usage of a virtual machine, the API offered on each platform can be vastly different. For example, virtual machines for mobile devices usually offer a much more limited API.

perspective. Especially with respect to virtual machine technology, research has boomed during the last decade. Efforts were made to better understand the behaviour of Java applications (e.g., [26, 33, 87]), and to improve the virtual machine (e.g., [30, 52]). There was renewed interest in garbage collection – which exists since the 1960's [77] – (e.g., [16, 21]), etc.

A cornerstone of experimental computer science, is the availability of benchmarks. First of all, benchmarks offer a way to validate new techniques, and to ascertain the (relative) performance enhancement said techniques make, compared to the state-of-the-art. Second, their existence allows the reproduction and verification of past experiments. As such, they are invaluable.

For a long time the only available industry standard Java benchmark suite was SPECjvm98 [103] and it was used by the majority of researchers. However, several of the SPECjvm98 benchmarks (*compress*, *db*, and *mpegaudio*) have been classified as *simple* benchmarks by Shuf et al. [99]. They argue that some SPECjvm98 benchmarks are not truly object-oriented and are thus not representative for real Java workloads.

To alleviate some of these issues other benchmark suites were built. In 2000 the Java Grande [28] benchmark suite was released, but it was seldom used in research papers. Also in 2000, SPECjbb [102] was released, providing researchers with a bigger workload than those offered by SPECjvm98. In 2005 a new version of SPECjbb was made available. More recently, the DaCapo suite [17] was released, more than doubling the number of available benchmarks. Furthermore, the DaCapo benchmarks were selected to have a larger memory footprint, and come with larger input sets compared to SPECjvm98. As such, they execute for a longer period of time compared to the SPECjvm98 benchmarks. Both the SPECjvm98 and DaCapo suites offer inputs to each benchmark in varying sizes, from very small to large. However, Java lacks benchmarks that can execute sufficiently long[6] such that a steady-state with respect to optimised methods is reached. To address this particular problem, both SPECjvm98 and DaCapo offer the possibility of executing a benchmark multiple times in a single virtual machine invocation. The idea is that the repeated processing of the same input set mimics steady-state behaviour. For a more elaborate description of the benchmark suites we use in this dissertation, we kindly refer to Appendix A.1. In 2001, 2002 and 2004, respectively, SPEC also released three versions of SPECjAppServer, which is a server-side Java benchmark mimicking a J2EE server. In the latest version, SPEC offers a true multi-tier system for benchmarking all major J2EE components. This benchmark was not used in this dissertation.

When something new comes along in computer science, inexorably people are interested in its performance. Obviously, if a computing platform does not offer decent performance, people will not adopt it. As such, new languages, platforms, etc., warrant a close performance inspection. Java definitely was no exception to this rule. Early on, people compared Java applications with

[6] Except for SPECjbb2000 and SPECjbb2005: they can run for as long as the user desires.

compiled (e.g., SPEC CPU) applications [33], focusing mainly on one obvious potential difference, i.e., branch behaviour. Others examined I-cache and D-cache behaviour and use mostly simulation to obtain results [87, 88]. Because Java application (generally) execute in a virtual execution environment, gaining insight in their behaviour and their performance issues is a complex matter. The virtual machine loads Java classes at runtime, handles thread scheduling, schedules its optimisation system in between regular application execution, and orchestrates garbage collection to ensure the heap does not fill up to the brim.

1.3 Three pitfalls in Java performance evaluation

We claim that Java performance evaluation is a hard problem to tackle, the reason being the complexity of the virtual machine, and the ubiquitous interaction between the virtual machine and the application.

At the onset of the research that ultimately lead to this dissertation, there were a number of important aspects of Java performance analysis that were either not well understood or largely ignored. In this dissertation, we will study three of these pitfalls and we provide solutions to circumvent them.

1. The performance of a Java application is the result of a complex interaction between the virtual machine, the application, and its input.

   Pitfall: The results obtained by performing experiments on one virtual machine are often not representative for another virtual machine, and extrapolating behaviour can lead to mistakes. The same applies to input sets: behaviour information acquired for one input set is not necessarily representative for another input set.

   Recommendation: if possible, employ multiple virtual machines during experimentation and be careful about selecting the application's input. Small inputs cause the virtual machine to have the most impact on the behaviour of the workload. With large inputs, resulting in longer running applications, the application mainly determines behaviour, although there always remains some impact of the virtual machine.

2. Java virtual machines exhibit non-determinism.

   Pitfall: all too often people ignore this fact or do not adequately deal with it when reporting performance.

   Recommendation: Use rigorous statistics to analyse performance.

3. Average application performance numbers can be useful, but they may yield no information that can help a programmer improve the performance of said application.

   Pitfall: you lose far too much information by considering average performance.

Solution: Exploit method-level phase behaviour to zoom into the application behaviour and to find bottlenecks.

In the following sections, we elaborate on these pitfalls.

1.4 Pitfall: Ignoring the interaction between the Java workload components

Prior to this work, people have been doing valuable research on characterising Java applications [26, 33, 57, 69, 70, 87, 88, 99]. However, they typically considered only one or two virtual machines in their methodology, as well as a single benchmark suite, typically SPECjvm98. Furthermore, simulation – which is slow – is often used to gather information. And, to limit the time spent, researchers restrict themselves to using a small input set, e.g., *s1* for SPECjvm98. However, therein lies a risk. Is it really so that a small input set – yielding a short running workload – is representative for a larger input set – yielding a longer running workload? Is the behaviour of a Java workload largely independent of the virtual machine on which it is executed, i.e., can conclusions made on one VM be transferred to another virtual machine? The answer to these question is mostly negative [43].

Basically, we can distinguish three important aspects that potentially have a large impact on the overall behaviour of a Java workload: the virtual machine executing the Java bytecode, the Java application itself and the input to the Java application.

Example 1.1. *Generally, programs obey the 80/20 rule, that is 80% of the time is spent in 20% of the – so-called hot – code, and Java programs are no exception to that rule. Hence, to achieve fast running programs the hot 20% should be optimised as much as possible. A production virtual machine will try to deliver on this promise. Still, each virtual machine implements its own optimisations. Moreover, a virtual machine may offer multiple optimisations levels. As such, the generated native code can differ quite a lot depending on the virtual machine used to run the Java application. Because optimised code is responsible for the major part of the execution, it follows naturally that the virtual machine plays a large role in determining the behaviour of a Java application. Intuitively, it is clear that a researcher cannot simply generalise results he*[7] *gets on a single virtual machine.*

Example 1.2. *The input of a program can have a profound impact on its behaviour. For example, the control flow of a program is steered by the input the program processes. In the case of multi-threaded applications, the execution of an application may vary across runs, even if the same input is provided to the program [48, 67], and if the side-effects of the program (e.g., the output) remain the same. Additionally, a*

[7] Throughout the dissertation, the male form of the pronoun is used but obviously the female form could be used just as well!

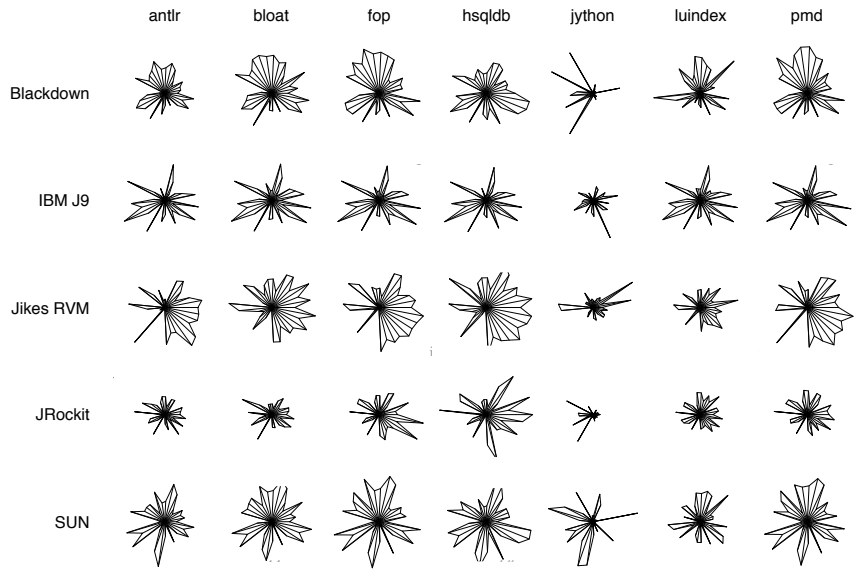


Figure 1.1: Kiviat diagrams of the DaCapo benchmarks executed on 5 different virtual machines using the *small* input set.

large input set will most likely cause more objects to be created, stressing the memory subsystem, and activating more garbage collections than a small input set.

Figures 1.1 and 1.2 show Kiviat diagrams of the execution characteristics[8] *measured on an AMD Athlon XP for several DaCapo benchmarks on several virtual machines. A kiviat diagram shows the data – normalised to* $[0; 1]$ *– on the radials of a unit-circle. Each radial represents a single characteristic and is cut off at the normalised value for that characteristic. The angle between the radials is obviously a function of the number of characteristics* n*, i.e.,* $\frac{2\pi}{n}$*. The intuition behind a kiviat diagram is to provide a human interpreter with a visual aid to distinguish (dis)similarities*[9]*. As can readily be observed in these figures, the size of the input changes which aspect contributes more to the behaviour. Clearly, in Figure 1.1, where the benchmarks were run with the small input set, the kiviat diagrams port most resemblance to each other when looking row-wise, i.e., per virtual machine. In Figure 1.2 on the other hand, where the benchmarks were run with the large input set, there is much more similarity column-wise, even though there still is some row-wise similarity.*

Each of the three aspects mentioned earlier – (i) virtual machine, (ii) application, and (iii) input, see Figure 1.3 – can thus have a large impact on the

[8] For details on these characteristics, see Chapter 2.

[9] Human minds are adept at recognising visual structures, such as faces.

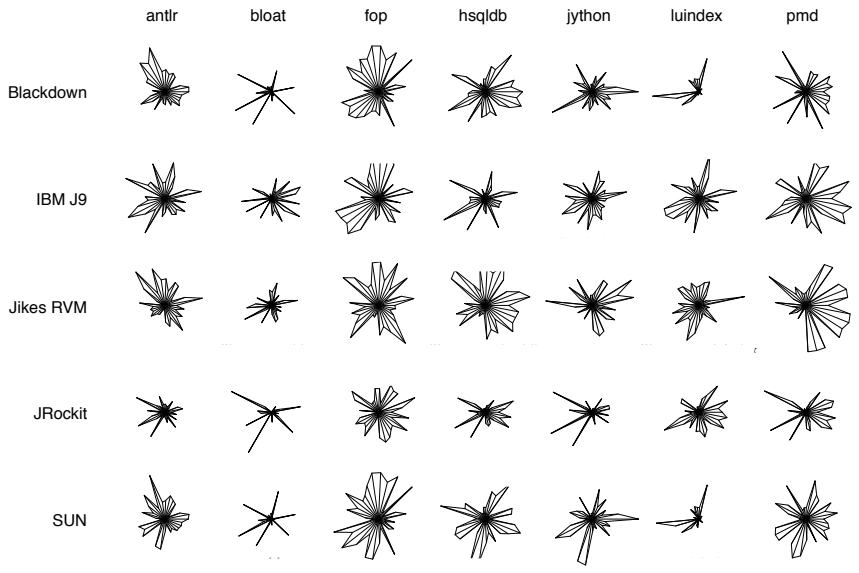


Figure 1.2: Kiviat diagrams of the DaCapo benchmarks executed on 5 different virtual machines using the *large* input set.

behaviour as observed at the micro-architectural level (in terms of branch behaviour, cache behaviour, instruction-level parallelism, etc.). The close interaction between the virtual machine, the Java application and the input is hard to understand, because it is difficult, if not outright impossible, to tease the aspects apart. The main question we seek to answer here is thus the following: how much of the behaviour as observed at the microprocessor level is due to the virtual machine, the Java application, and the input to the application? Is there a single aspect that primarily governs the execution behaviour? If not, how large is the impact of the Java application? And what is the impact of the input to the application?

In our first contribution, we answer the above questions by applying the following methodology. For a number of benchmarks and virtual machines, we measure a set of execution characteristics at the microprocessor level using hardware performance counters. Subsequently we conduct a thorough statistical analysis to gain better insight in the interaction between the virtual machine, the application and its input.

Our analysis shows that for small input sets, the main influence on the behaviour stems from the virtual machine. For larger inputs, the application plays a more prominent role in determining behaviour. Basically, the longer

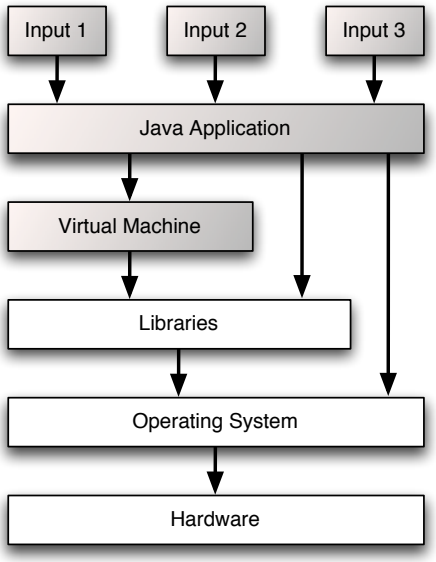


Figure 1.3: Key aspects monitored for gauging their relative influence on the execution observed at the lowest level.

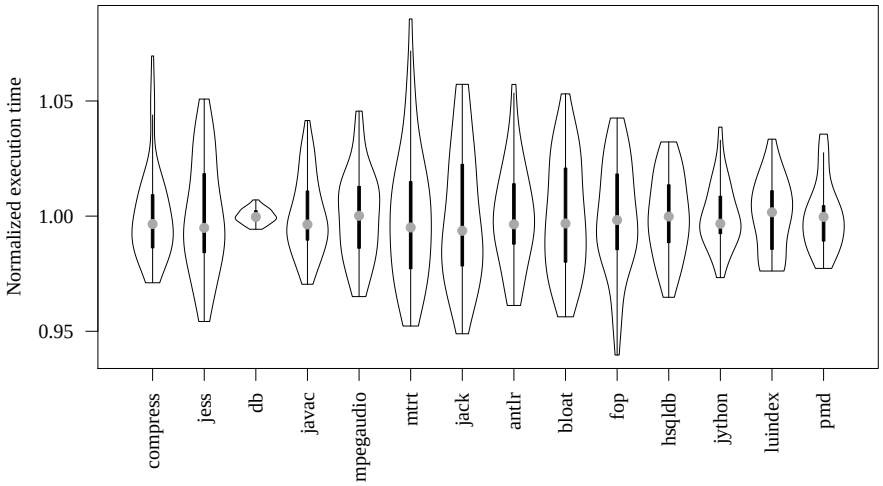
an application runs, the more it contributes to the overall behaviour, and the lesser the virtual machine contributes. Of course, as indicated in Example 1.1, the combination of both factors plays a part as well. These results were first described in:

How Java Programs Interact with Virtual Machines at the Micro-architectural Level. *Lieven Eeckhout*, *Andy Georges*, and *Koen De Bosschere*. In Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), pages 169–186, 2003.

1.5 Pitfall: Poor data analysis

The behaviour of Java applications is non-deterministic; this is due to several virtual machine components such as timer-based sampling for driving the optimisation system, garbage collection, thread scheduling etc. For example, if a method is optimised sooner, rather than later, the overall execution time will most likely be smaller. Consequently, we end up with variability in execution time. This effect is quantified in Figure 1.4, which shows the nor-

(a) first benchmark iteration



(b) tenth benchmark iteration

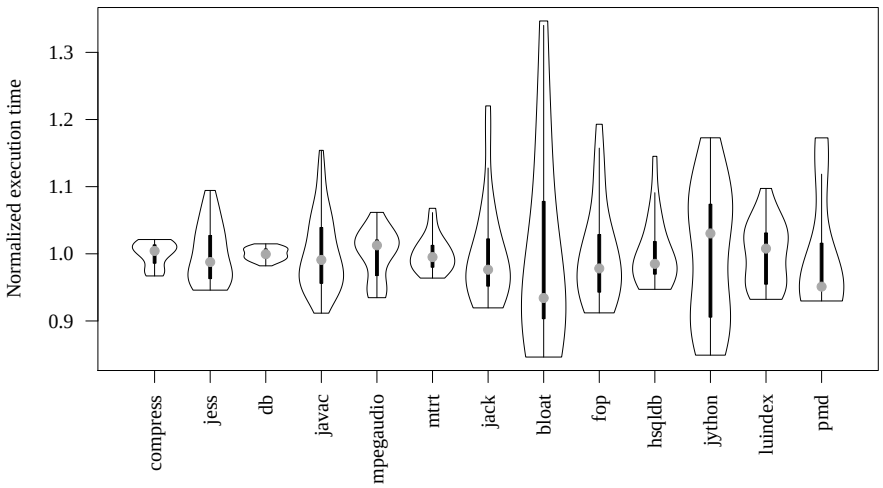


Figure 1.4: Variability observed in the execution of the SPECjvm98 (using the *s100* input set) and DaCapo (using the *default* input set) benchmarks on the AMD Athlon platform, using Jikes RVM. For each benchmark, the execution time is shown for (a) 30 invocations with a single benchmark iteration (one-run or start-up performance) and (b) 10 invocations measuring the tenth benchmark iteration (steady-state performance).

malised execution time for experiments in which each benchmark is executed thirty times on the Jikes RVM with a GenMS garbage collector on an AMD Athlon XP machine[10]. For each benchmark, the heap size was fixed to twice the minimal heap size required to allow executing the benchmark to completion. The graph in the figure shows violin plots [56]; all values are normalised to a mean of one. In addition to the information conveyed in a regular box plot, the shape of a violin plot represents the density. The middle point shows the median; the thick vertical line represents the first and third quartiles (50% of all the data points are between the first and third quartile); the thin vertical line represents the upper and lower adjacent values (representing an extreme data point or 1.5 times the interquartile range from the median, whichever comes first); and the top and bottom values show the maximum and minimum values.

We can immediately make several interesting observations from this graph. First, run-time variability can be fairly significant. For most of the benchmarks, the coefficient of variation (CoV), defined as the standard deviation divided by the mean, is around 2% and is higher for several benchmarks. Second, the maximum performance difference between the maximum and minimum performance number varies across the benchmarks, and is generally around 8% for start-up. Third, most of the violin plots in Figure 1.4 show that the measurement data is approximately Gaussian distributed with the bulk weight around the mean. A more rigorous statistical test, such as the Kolmogorov-Smirnov test, does not reject the hypothesis that for the experiments shown above, the data is approximately Gaussian distributed.

When researchers try to quantify Java performance, they adopt a wide range of evaluation methodologies and performance metrics. A survey of 50 papers published recently at premier conferences, such as OOPSLA, PLDI, VEE and CGO, yields the following results.

About one third of the papers (16 out of 50) do not specify any methodology regarding their setup or data analysis they use to obtain their results. Among the other papers, there is no consensus whatsoever on what strategy to follow, especially regarding the data analysis used to report results. Example data analysis methodologies include: (i) best of k *runs*, (ii) second best of k *runs*, (iii) median of k *runs*, (iv) mean of k *runs*, and (v) in 2 cases a confidence interval was reported. We present the survey results in full detail in Chapter 3, but it will be immediately clear to the reader that it is quite hard to compare results across papers if a different approach is used.

More importantly, the prevalent approaches used can lead to misleading or incorrect results, as demonstrated by the following example.

Example 1.3. *Consider the following situation. A researcher wishes to see how various garbage collection strategies perform on his application of interest, in this case the db benchmark. He uses the Jikes RVM, for which he builds several configura-*

[10]We consider a single benchmark iteration per VM invocation. We will later refer to this experimental design setup as *start-up* performance

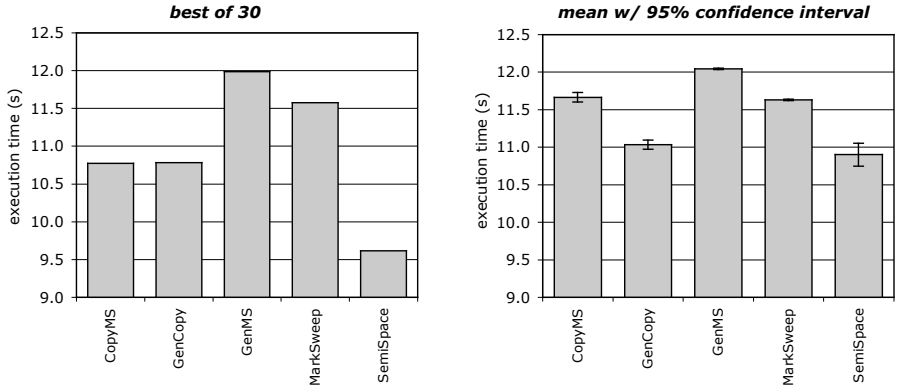


Figure 1.5: An example illustrating the pitfall of using a prevalent Java performance data analysis method: on the left the best-of method seems to indicate that the SemiSpace garbage collector outperforms the GenCopy collector. However, the right graph, showing a rigorous statistical analysis of the same data shows there is no noticeable difference between both collectors. The experiment was conducted on an AMD Athlon machine using the Jikes RVM on the *db* benchmark with a 120MB heap size.

tions, each for a particular GC strategy: CopyMS, Gencopy, GenMS, MarkSweep and SemiSpace. For details regarding these GCs, we refer to [16]. The researcher is aware that there may be some variation on his measurements, so he confiscates a machine from the lab and makes sure it is idle but for the Jikes RVM running on it, and fires up 30 experiments using a heap size of 120MB. He then uses a best-of approach and finds that, by and large, the SemiSpace collector performs best for this application (shortest execution time, see left graph in Figure 1.5).

However, he is clearly mistaken. Had he looked a bit more carefully, he would have noticed that in a only few of the 30 experiments, SemiSpace did very well. Applying a rigorous statistical analysis, that constructs a confidence interval for each of the GC strategies, he would arrive at a very different conclusion. Essentially, there is no statistically significant performance difference between GenCopy and SemiSpace for this experiment, as can be observed in the right graph in Figure 1.5.

Basically, in the above example, there are 10 possible pairwise GC comparisons. For three of them (i) GenCopy vs. SemiSpace, (ii) CopyMS vs. GenCopy, and (iii) CopyMS vs. MarkSweep, the prevalent approach reaches a different conclusion from the rigorous approach. This means that in 33% of the comparisons, we would be wrong. For one, based on the best method, one would conclude that SemiSpace clearly outperforms GenCopy, as indicated in the example. The reality though is that the confidence intervals for both garbage collectors overlap and, as a result, the performance difference seen between both garbage collectors is likely due to the random performance variations in the system under measurement, which is confirmed by a Student

t-test. In fact, we observe a large performance variation for SemiSpace, and at least one really good run along with a large number of less impressive runs. The best method reports the really good run whereas a statistically rigorous approach reliably reports that the average scores for GenCopy and SemiSpace are very close to each other. Second and similarly, based on the best method, one would conclude that the performance for the CopyMS and GenCopy collectors is about the same. However, the statistically rigorous method shows that GenCopy significantly outperforms CopyMS. A third case where the wrong conclusion will be made when using the best method is the comparison of CopyMS and MarkSweep. Based on a statistical rigorous data analysis (which in our opinion reflects reality best), there is no significant difference between both garbage collection strategies for this particular benchmark with the given heap size.

In our second contribution, we advocate a rigorous statistical approach to deal with non-determinism in the measurements. Key to our approach is the use of confidence intervals. However, depending on the experiments a researcher wishes to perform, there are various approaches he can take, e.g., the analysis will be somewhat different if a single factor is varied, from a setup where multiple factors are taken into account when assessing performance. We propose a sound approach for dealing with both start-up and steady-state performance. In the former case, it suffices to run enough virtual machine invocations. In the latter case, it is also paramount that in each virtual machine invocation, the benchmark is iterated ample times.

We thoroughly compare the prevalent methodologies, both for start-up and steady-state against a statistically rigorous data analysis. The use-case we examine is the performance comparison of 5 different garbage collectors in the Jikes RVM MMTk [16] using 14 benchmarks from the SPECjvm98 and DaCapo benchmark suites. For this experiment, we show that a significant number of conclusions based on prevalent-based comparisons are misleading or incorrect. We do not claim that of the papers published in the past the results are in fact wrong, but there is no way to find out without doing the experiments over. Clearly, this shows that there is a need for a rigorous and sound data analysis when evaluating Java performance.

This work was published in:

Statically Rigorous Java Performance Evaluation. *Andy Georges*, *Dries Buytaert*, and *Lieven Eeckhout*. In Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), pages 553–568, 2007.

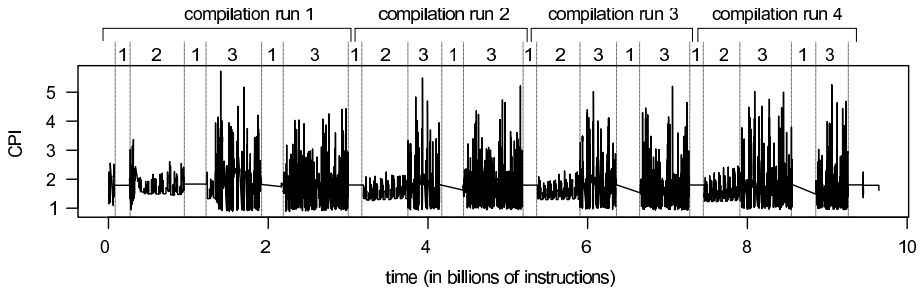


Figure 1.6: The graph shows the CPI sequence of *javac*, executed on the Jikes RVM using the *s100* input set.

1.6 Pitfall: Average performance provides little information to the programmer

As we have seen, there is a complex interaction between all parties contributing to the execution of a Java application. Furthermore, the applications themselves are growing in complexity. Hence, understanding performance behaviour of an application is far from trivial. Worse, applications typically exhibit varying behaviour over time. As such, the global performance, e.g., average CPI of an application, gives little information to the programmer with respect to bottlenecks that may be present.

Example 1.4. *Figure 1.6 shows javac's CPI evolution as a function of time. The CPI is plotted for each invocation of the selected methods. We can clearly distinguish a number of high-level recurring phases, where the pattern looks very similar. We also see that a number of methods have a CPI that is much worse, i.e., higher, than the average CPI – which is 1.67. If a programmer wants to improve the performance of his application, the high-CPI methods that are executed frequently are the methods of interest.*

In our third contribution, we show that the execution of a Java program exhibits phase behaviour at the method-level. This means that throughout the execution a number of recurrent phases are present – a phase is then a set of segments in the program execution that exhibit similar behaviour – and these phases correspond to program methods. This means that a phase actually corresponds to a part of the dynamic call tree. The root of such a tree is the method that identifies the phase. The underlying assumption of method-level phase behaviour is that phases of execution correspond to the code that gets executed. In particular, different methods are likely to result in dissimilar behaviour and different invocations of the same method are likely to result in similar behaviour. Hence, there is less variability within phases than between phases.

An application of this concept is finding performance bottlenecks in a Java

program. For each detected phase, or method, we can assemble performance metrics, such as CPI, cache misses, etc. This makes it possible to correlate bad performance – e.g., CPI values that are way above average – with source code.

This work was published in:

Method-level Phase Behaviour in Java Workloads. *Andy Georges*, *Dries Buytaert*, *Lieven Eeckhout*, and *Koen De Bosschere*. In Proceedings of the 19th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA), pages 270–287, 2004.[11]

1.7 Other work not incorporated in this dissertation

In addition to the work we describe in this thesis, we made the following contributions to the scientific community.

1.7.1 Record-replay for Java

We developed a portable record/replay system for Java, called JaRec. It correctly replays multi-threaded Java applications that are free of data races. To do so, JaRec records the order of synchronisation operations and faithfully replays them in the same order during replay to the application. The entire infrastructure is built in Java. Provided that the Java virtual machine allows interaction using the Java Virtual Machine Profile Interface (JVMPI), JaRec usage requires no modification to the virtual machine. If the JVMPI is not available, only small modifications to the virtual machine are required. On systems with limited memory resources, it is possible to run JaRec in a distributed fashion. This makes JaRec a feasible tool for debugging multi-threaded Java applications on an embedded system.

JaRec: A Portable Record/Replay Environment for Multi-Threaded Java Applications. *Andy Georges*, *Mark Christiaens*, *Michiel Ronsse*, and *Koen De Bosschere*. In Software - Practice and Experience. John Wiley & Sons, Ltd. Vol. 34, pages 523–547, 2004.

1.7.2 A comparison between Java and classical workloads

Because Java applications are usually running on top of a Java virtual machine, and are written in an object-oriented language, Java workloads poten-

[11] This work is detailed in Chapter 6; it describes joint work with Dries Buytaert. The main contribution of Dries consists of the vertical instrumentation aspects of the methodology, where my focus was on the analysis. As such, this dissertation describes my contributions to this work.

tially exhibit different execution characteristics from compiled C and Fortran workloads. We make a thorough comparison between both workload types using the performance counters present on an AMD Duron platform and a rigorous statistical data analysis. Our setup uses multiple virtual machines and benchmarks from the SPECjvm98, SPECjbb2000 and Java Grande suites. We show that Java workloads are significantly different from SPEC CPU. Java workloads have significantly less L1 data cache misses, significantly more L2 instruction TLB misses and significantly more mispredicted function returns. The reason for having fewer data cache misses is the better data locality in Java applications. The reason for having more L2 instruction TLB misses is mainly the fact that throughout the execution, (re)compiled and optimised code is placed on pages that are accessed as data before the execution moves there. Moreover, we show that the execution characteristics for which both workload types differ are subjective to the virtual machine used.

Comparing Low-Level Behavior of SPEC CPU and Java Workloads. *Andy Georges*, *Lieven Eeckhout*, and *Koen De Bosschere*. In Proceedings of the Advances in Computer Systems Architecture: 10th Asia-Pacific Conference, ACSAC 2005. Springer-Verlag. Lecture Notes in Computer Science. Vol. 3740, pages 669–679, 2005.

1.7.3 Performance prediction for classical workloads

A key challenge in benchmarking is to predict the performance of an application of interest on a number of platforms in order to determine which platform yields the best performance. We measure a number of micro-architecture-independent characteristics from the application of interest, and relate these characteristics to the characteristics of the programs from a previously profiled benchmark suite. Based on the similarity of the application of interest with programs in the benchmark suite, we make a performance prediction of the application of interest. We propose and evaluate three approaches (normalisation, principal components analysis and genetic algorithm) to transform the raw data set of micro-architecture-independent characteristics into a benchmark space in which the relative distance is a measure for the relative performance differences. We evaluate our approach using all of the SPEC CPU2000 benchmarks and real hardware performance numbers from the SPEC website. Our framework estimates per-benchmark machine ranks with a 0.89 average and a 0.80 worst case rank correlation coefficient.

Performance Prediction Based on Inherent Program Similarity. *Kenneth Hoste*, *Aashish Phansalkar*, *Lieven Eeckhout*, *Andy Georges*, *Lizy K. John*, and *Koen De Bosschere*. In Proceedings of the Fifteenth International Conference on Parallel Architectures and Compilation Techniques (PACT), pages 114–122, 2006.

1.7.4 HPM sampling for dynamic compilation

All high-performance production JVMs employ an adaptive strategy for program execution. Methods are either first interpreted or executed unoptimised and then an on-line profiling mechanism is used to find a subset of methods that should be optimised during the same execution. We empirically evaluate the design space of several profilers for dynamic compilation and show that existing online profiling schemes suffer from several limitations. They provide an insufficient number of samples, are untimely, and have limited accuracy at determining the frequently executed methods. We describe and comprehensively evaluate HPM-sampling, a simple but effective profiling scheme for finding optimisation candidates using hardware performance monitors (HPMs) that addresses the aforementioned limitations. We show that HPM-sampling is more accurate, has low overhead, and improves performance by 5.7% on average and up to 18.3% for the SPECjvm98 and DaCapo benchmarks when compared to the default system in Jikes RVM.

Using HPM-Sampling to Drive Dynamic Compilation. *Dries Buytaert*, *Andy Georges*, *Michael Hind*, *Matthew Arnold*, *Lieven Eeckhout*, and *Koen De Bosschere*. In Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), pages 553–568, 2007.

1.8 Terminology used in this dissertation

In this dissertation, we use several terms, that we define specifically in the context of this work.

Workload. In the literature, the term *workload* can mean several things. The Oxford dictionary defines it as *the amount of work to be done by someone or something*. In the context of computer architecture it usually means a number of applications that have to be run either sequentially or simultaneously on a computer. In this dissertation, we take a workload to mean the following: a given set of instructions to be executed by the CPU. More specifically, in the context of executing Java applications, a workload is the resulting set of instructions executed when a virtual machine runs a Java application with a certain input. When we use the term in combination with a benchmark or virtual machine name, e.g., *mpegaudio* workload, we mean the resulting set of instructions when executing the *mpegaudio* benchmark.

VM invocation. By this term, we mean the virtual machine process itself. Thus, a VM *invocation* means starting the virtual machine and having it run a certain Java application. After the application has finished, the VM exits.

Benchmark iteration. Both SPECjvm98 and DaCapo offer an execution harness which executes the actual benchmarks one or more times, without restarting the virtual machine, i.e., in a single invocation the virtual machine may execute multiple benchmark *iterations*. Because the virtual machine is not restarted in between iterations, the optimiser and compiler will have optimised most methods during the first few iterations.

Start-up execution. Typically, Java application start-up execution means the initial execution part of this application in a single VM invocation, where the majority of method compilation takes place. When referring to execution inside the SPECjvm98 or DaCapo harnesses, as is the case in this dissertation, *start-up* means the first iteration of the executed benchmark.

Steady-state execution. Generally, steady-state refers to the execution of an application that has been running for a sufficiently long time, such that the virtual machine has optimised the most frequently executed methods. It is notoriously difficult to decide when *steady-state* has been reached. When an application is running for a longer period of time, the optimisation mechanism might have collected enough samples of an unoptimised methods to assign a sufficient hotness level to new methods, or lift already optimised methods over a higher threshold and proposing them for further optimisation.

1.9 Managed runtime environments in general

This dissertation places the emphasis on the Java platform. However, the pitfalls can also be found in other managed runtime environments or virtual execution environments (VEEs). Consequently, the solutions we present are applicable in a broader context than Java. Virtualisation and managed runtime systems are here to stay, and the concept is increasingly gaining popularity. A number of languages that are quite hot these days – and thus are gaining importance – use managed runtime systems to execute the programs. PHP, Python, C#, Ruby, etc. immediately come to mind. Additionally, several compiled languages include a runtime system, e.g., Haskell. Typically the environment for these languages is quite similar to the Java platform. Garbage collection, runtime optimisation, and thread scheduling are commonplace. Furthermore, virtualisation is being extended to complete systems, e.g., VMWare, Xen, etc. We conjecture that performance analysis in these environments suffers from similar pitfalls. Therefore, we advise researchers and benchmarkers to take the arguments presented in this work to heart when conducting experiments in a managed runtime environment.

Of course, good benchmarking practice is applicable to all environments, classical compiled languages such as C, C++ and Fortran not excluded. In particular, the techniques presented in Chapter 4 can also be applied to ap-

plications written in these languages. However, due to the lack of significant sources of non-determinism, (single-threaded) programs written in these languages are less subject to error.

1.10 Overview

This dissertation is organised as follows. In Chapter 2, we show that interaction effects between the virtual machine and benchmarks are omnipresent, and that care should be taken when drawing conclusions based on experiments with a small input set, or with only one or two virtual machines. In Chapter 3, we detail prevalent performance analysis approaches, and in Chapter 4, we show that these approaches suffer several shortcomings, causing them to lead to either misleading or incorrect conclusions. In Chapter 5, we revisit replay-compilation – a prevalent experimental design methodology – and suggest improvements to current practice. In Chapter 6, we provide a phase-based technique to gain more insight into the behaviour of a Java application, separating it from (most of) the virtual machine components. Finally, in Chapter 7, we briefly present our main conclusions and touch upon possible routes to be explored in future work.

Chapter 2

Interaction

A little rudeness and disrespect can elevate a meaningless interaction to a battle of wills and add drama to an otherwise dull day. – **Calvin**

The main insight provided in this chapter is that Java performance is the result of a complex interaction between the virtual machine and the application. More specifically, if the goal is to analyse the performance of a Java application, it is paramount that the application executes for a sufficiently long period of time, otherwise one will measure the execution of the virtual machine, rather than the application. In this chapter we discuss this point and its ramifications.

2.1 Introduction

Essentially, we distinguish three important components that impact the overall behaviour of the Java workload: (i) the virtual machine, (ii) the Java application, and (iii) the input to the application. The latter essentially is what makes the application tick, so its importance should not be underestimated, as we will show. Due to the write-once, run (almost) everywhere nature of the Java platform, the virtual machine is another important component. It directs the interpretation, the compilation and subsequent optimisation, the memory allocation and reclamation (garbage collection), the thread scheduling[1], etc. The execution characteristics of the complete workload will be significantly influenced by the code generation part of the virtual machine. The garbage

[1]More in particular, a VM may implement an m-to-n threading system, where m Java threads are mapped onto $n \leq m$ OS threads. For example, Jikes RVM uses an m-to-1 system when running on a single core CPU. HotSpot on the other hand maps all Java threads to their own OS thread, effectively letting the OS handle thread scheduling.

collection strategy will have an immediate effect on the execution time, and thus on perceived performance. Finally, the application also has a profound impact on the behaviour – it is to be expected that a database application behaves differently from a game application.

We are notably interested in the behaviour observed at the micro-architectural level, as this gives us an idea of the real performance of a workload. Each of the components mentioned above can influence the behaviour at this execution level. Additionally, events occurring in the microprocessor can influence the software layer. For example, a cache miss in the level 2 cache can delay the execution such that the sample taken by the virtual machine to make an optimisation decision differs from the sample that would have been taken had the miss not occurred. The main barrier for gaining insight into the impact that various components have on overall performance and the way in which they affect each other, is the very complex interaction they have. Thus, a first step towards understanding is to tease apart the interaction effects.

The main question we want to address in this chapter is thus the following: how much of the behaviour as observed at the microprocessor level is due to the virtual machine, the Java application, and the input to the application? For example, most virtual machines currently employ a JIT optimising compilation strategy. But how big is the impact of the actual implementation of the JIT engine on the observed behaviour? I.e., do virtual machines implementing more or less the same strategy behave similarly? Secondly, how large is the impact of the Java application? Is the behaviour of a Java workload primarily determined by the Java application or by the virtual machine? And what is the impact of the input to the Java application? The answer to these questions will unveil the very pitfall we are addressing in this chapter.

Prior to this work, researchers typically considered only one (sometimes two) virtual machines [87, 88]. They focused mostly on the SPECjvm98 benchmark suite, as it was about the only standard suite available. Additionally, several studies use a small input set, e.g., *s1* with the SPECjvm98 suite, especially when simulation was used for characterising behaviour because of the large simulation times. Therefore, we deem the following questions need to be answered. Is such a methodology reliable for Java workloads? What happens if the behaviour of a Java workload is highly dependent on the chosen virtual machine? Can we translate conclusions made for one virtual machine to another virtual machine? Also, is SPECjvm98 representative of other Java applications? I.e., are the conclusions taken based on SPECjvm98 valid for other Java programs? And is using a small input, e.g., SPECjvm98 *s1*, yielding a short-running Java workload representative for a large input, e.g., *s100*, yielding a long-running Java workload?

The answers to the above questions are of interest for several research domains. First, Java developers can learn how their code behaves on the microprocessor and how it interacts with the virtual machine. For example, if the overall behaviour of the workload is primarily influenced by the virtual

machine instead of the Java applications, time can better be spent on improving robustness and re-usability instead of performance tweaking the application. Second, virtual machine developers will get more insight into which behavioural aspects are influenced by the virtual machine implementation and into the synergy that exists between the virtual machine and the application. Third, for microprocessor designers relying on time-consuming simulations it is extremely useful to know whether small inputs result in similar behaviour as large inputs and can thus be used to reduce the total simulation time without compromising the accuracy of their simulation runs [44].

In the remainder of this chapter, we discuss the following items. We briefly introduce the methodology employed to give answer to the aforementioned questions. Then we discuss the experimental setup. Subsequently, we detail the statistical techniques used to increase our understanding. Next, we uncover the pitfall in the evaluation section. Finally, we briefly describe related work, and conclude.

2.2 Methodology: overview

To address the questions raised in the previous section, we measure the performance at the heart of the computer, i.e., on the microprocessor. We employ the methodology that was first used in [43, 44, 45]. We first give a general overview of the methodology; the individual steps are discussed in more detail in subsequent sections.

We use the hardware performance counters that are available on modern CPUs to obtain information about 33 execution characteristics of interest for a number of Java applications from the SPECjvm98 [103] and DaCapo [17] suites. Our study is comprised of multiple virtual machines, see Table 2.1. The complete setup is discussed in detail in Section 2.3. Important to know is that we use both a small input set size and a large input set size. The idea is that the former causes the application to complete in a short(er) period of time; the latter causes the application to run for a longer period of time. This experiment yields a large set of data, which can be organised as a 33-column matrix with one row for each workload or (virtual machine, benchmark, input) tuple: the rows form the cases, and the columns form the characteristics. One can easily imagine these characteristics to span a 33-dimensional space, in which the cases reside. In this space – referred to as the workload space – workloads that show similar behaviour across (most of) these 33 characteristics can be found close together, whereas workloads that differ significantly will lie further apart.

Most human minds have difficulty coping with a three or four dimensional space, let alone a 33-dimensional one[2]. Moreover, not all execution character-

[2]Even a fairly comprehensible graph such as a kiviat diagram as shown in Section 1.4 is hard to parse when a lot of dimensions are involved.

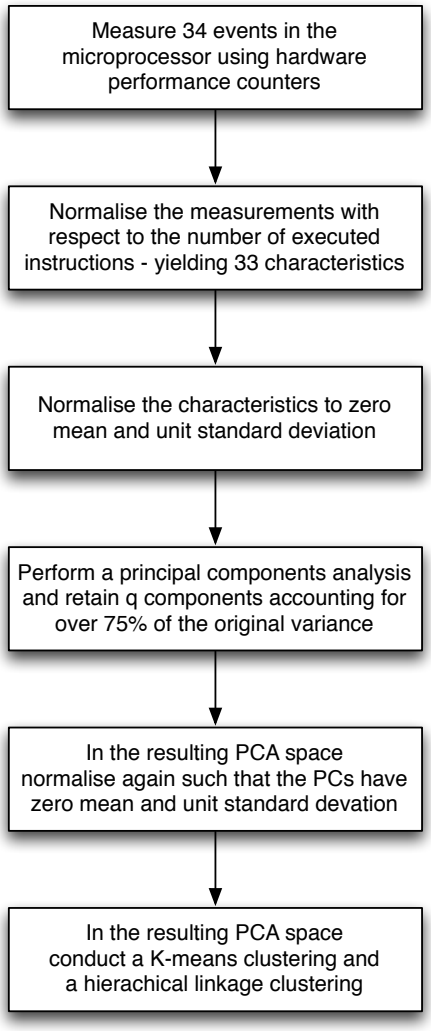


Figure 2.1: Overview of the methodology.

istics are independent. For example, from the experiments described later in this chapter, we find that the CPI show a correlation of 0.92 with the number of data cache reloads from main memory per instruction. To further complicate matters, it is difficult to find similarities or difference in this space, because the dimensions exhibit different variability. This means that a simple intuitive similarity measure such as Euclidean distance cannot be immediately applied in the workload space. Indeed, for non-normalised data, the contribution to the distance is larger for dimensions with a higher variance. Through normal-

ising all characteristics to have zero mean and unit variance, all dimensions (in this case, the events measured) make equal contributions. To enhance understanding, we reduce the complexity using Principal Components Analysis (PCA) [62]. Briefly put, PCA is a multivariate statistical data transformation technique that is aimed at mapping high-dimensional data onto a small-dimensional space with little loss in information, which allows it to increase the understandability. The basic idea is that a linear transformation yields a new basis in the workload space, transforming it into a so-called PCA space. This basis is chosen as such that the first dimensions account for most of the variance in the workload space. Thus, it is possible to drop higher dimensions in the PCA space without losing too much information from the data set. An additional advantage is the fact that the basis in PCA space consists of uncorrelated vectors, and that the Euclidean distance becomes an acceptable distance metric. Thus, the human evaluator gains a better understanding due to two reasons: (i) lower dimensionality, and (ii) lack of correlation between dimensions. In our experiments, we usually retain eight dimensions out of 33, accounting for approximately 75% of the variance that is present in the original data.

In a third step we use Cluster Analysis (CA) [62] to present the workload similarity in an even more intelligible fashion. CA arranges the workloads according to similarity, i.e., similar workloads are closely linked. Prior to CA, we renormalise the principal components, to give each PC the same weight in the subsequent analysis. We will iteratively link the workloads from the (normalised) PCA-space together, such that similar workloads are linked together earlier. It follows immediately that, should the workloads be clustered by virtual machine, the VM will be the main influence on the behaviour. If, on the other hand, the workloads are clustered together by application, we will conclude that the Java application mostly determines overall execution behaviour. Finally, if multiple inputs for the same benchmark are clustered together, this means that the input set has a limited effect on overall performance, and therefore conclusions from small input sets can be extrapolated to large input sets.

2.3 Experimental setup

We use five virtual machine configurations in these experiments, listed in Table 2.1. Two of them implement the Java 1.5 specification (HotSpot and J9), the others implement the Java 1.4 specification, although Jikes RVM is in the process of being converted to the Java 1.5 specification.

In this study, we use 14 benchmarks, 7 from SPECjvm98 [103], and 7 from DaCapo [17], see Table 2.2. From the latter suite, we only use the benchmarks that run to completion on each of the virtual machines listed in Table 2.1. For each benchmark, we use a single fixed heap size. We determined the minimal heap size required to run the benchmark to completion on every virtual ma-

Virtual machine	Configuration
SUN JRE 1.5	Hotspot with the client JIT compiler (mixed-mode), generational stop-the-world GC with a nursery, a tenure space and a permanent space
Blackdown JRE 1.4.1	Hotspot with the client JIT compiler (mixed-mode), generational GC with a nursery and a tenure space
Jikes RVM	adaptive optimising compilation-only mode, generational mark-sweep GC (GenMS)
JRockit 1.4.1	Adaptive optimising mode, generational copying GC
IBM J9	Interpretation plus a JIT compiler (mixed-mode), non-generational mark-sweep compacting GC

Table 2.1: Java virtual machines used to study the interaction between the VM and the Java application.

Benchmark	Description	Heap size used (MB)
compress	file compression	48
jess	puzzle solving	32
db	database	64
javac	Java compiler	64
mpegaudio	MPEG decompression	32
mtrt	raytracing	48
jack	parsing	48
antlr	parsing	64
bloat	Java bytecode optimisation	112
fop	PDF generation from XSL-FO	112
hsqldb	database	352
jython	Python interpreter	144
luindex	document indexing	64
pmd	Java class analysis	128

Table 2.2: The SPECjvm98 (top seven) and DaCapo (bottom seven) benchmarks used to study the interaction between the VM and the Java application. The rightmost column indicates the (fixed) heap size we use for the experiments described in this chapter. It is twice as large as the minimal heap size required to run the benchmarks to completion on every virtual machine used in these experiments.

chine and we use a heap size fixed at twice that amount. For this study, we consider start-up execution, i.e., we iterate the benchmarks only once in each virtual machine invocation.

The hardware platform we use for the experiments described in this chapter consists of an AMD Athlon XP machine. The CPU is clocked at 2.1GHz, and has a 512MB L2 on-chip cache. The computer has 2GB of main memory, and runs the Linux 2.6.18 operating system. The measurements were taken on an otherwise idle machine, i.e., an unloaded machine in multi-user mode (with only a single user on the machine) but with networking, etc. turned on.

2.3.1 Execution characteristics measured with hardware performance counters

Modern microprocessors commonly contain hardware performance counters. For example, the AMD Athlon XP is equipped with four performance counter registers that can be used to obtain information on the CPU's usage during the execution of a program. As such, they naturally characterise the execution behaviour of a Java application. It can be argued that using hardware performance counters yields several benefits over alternative characterisation techniques, such as simulation or instrumentation. First of all, measurement is done directly on the bare hardware, which means it can proceed at the speed of the native execution of a program. Both simulation and instrumentation are less efficient, as they result in a serious slowdown, varying from 1000 times up to 100,000 times compared to native execution [7]. The advantage of simulation on the other hand is that one can easily vary the cache, branch predictor, ISA, etc. Second, there is no perturbation, as can happen when using instrumentation. Third, measuring kernel activity comes for free. Other techniques require either instrumenting the operating system kernel, or using a full-system simulator, adding more perturbation or slowdown. This is an important consideration given the fact that Java workloads spend up to 10% of their execution in the kernel [69]. Finally, measuring on real hardware discards the shortcoming of a software simulation model, which can lead to inaccuracies due to its higher abstraction level [36].

Unfortunately, using hardware performance counters also raises some concerns. The main concern is that there is a slight impreciseness, due to non-determinism. This means there can be some variation when measuring the same execution twice. This can be due to, for example, cache contention due to multitasking, interrupts, etc.[3] Additionally, as we noted in the introductory chapter, timer-based sampling may result in even more variability. Furthermore, the number of available hardware counters is much smaller than the number of events that can be measured. Limiting ourselves to 34 interesting events, listed in Table 2.3, requires us to execute each workload at least nine times to obtain the information of interest. If we follow the procedure outlined

[3] This is a hardware issue, not a software problem.

in Chapter 4 – we measure each workload 30 times – this measurement takes a factor 270 longer than a single execution at native speed.

To allow access to the hardware performance counters, the Linux kernel has been patched with the *perfctr*[4] patch. We used version 2.6.19 for the experiments described in this chapter.

2.3.2 Workload characteristics

The processor events that we measure to characterise the Java workload execution are tabulated in Table 2.3. These 34 events can be roughly divided in six groups:

- **General characteristics**. This group of events contains the number of clock cycles needed to execute the application; the number of retired x86 instructions; the number of retired operations – x86 instructions are broken down to fixed-length and much simpler operations; the number of retired branches, etc.
- **Processor front-end**. Here we have grouped characteristics that are related to the processor front-end, i.e., the I-cache and the fetch unit: the number of fetches from the L1 I-cache, the number of L1 I-cache misses, the number of instruction fetches from the L2 instruction cache and the number of instruction fetches from main memory. Next to these characteristics, we also measure the L1 I-TLB misses that hit the L2 TLB, as well as the L1 I-TLB misses that also miss the L2 I-TLB. In addition, we also measure the number of fetch unit stall cycles.
- **Branch prediction**. This group measures the performance of the branch prediction hardware: the number of branch taken/not-taken mispredictions, the number of branch target mispredictions, the number of the return address stack (RAS) hits, etc.
- **Processor core**. The performance counters that deal with the processor core basically measure stall cycles, i.e., cycles in which no new instructions can be further pushed down the pipeline due to data, control or structural hazards, for example, due to a read-after-write dependency, an unavailable functional unit, an unresolved D-cache miss, a branch misprediction, etc. In this group we make a distinction between the following events: an integer control unit (ICU) full stall, a reservation station full stall, a floating-point unit (FPU) full stall, load-store unit queue full stalls, and a dispatch stall which can be the result of a number of combined stall events.
- **Data cache**. We distinguish the following characteristics related to the data cache: the number of L1 D-cache accesses, the number of L1 D-cache misses, the number of refills from L2, the number of refills from

[4] `http://user.it.uu.se/~mikpe/linux/perfctr/`

Component	Description
general	clock cycles retired x86 instructions retired operations retired branches retired taken branches retired far control instructions retired near return instructions
processor front-end	L1 I-cache fetches L1 I-cache misses L2 instruction fetches instruction fetches from memory L1 I-TLB misses, that hit the L2 I-TLB L1 and L2 I-TLB misses fetch unit stall cycles
branch prediction	retired mispredicted branches retired mispredicted taken branches retired mispredicted near return instructions mispredicted branches due to address miscompare return address stack hits return address stack overflows
processor core	dispatch stall cycles (combined events) integer control unit full stall cycles reservation station full stall cycles floating-point unit full stall cycles L1 D-cache load-store unit full stall cycles L2 cache/memory load-store unit full stall cycles
data cache	L1 data cache accesses (equals load-store operations) L1 data cache misses refills from the L2 cache refills from main memory writebacks L1 D-TLB misses, that hit the L2 D-TLB L1 and L2 D-TLB misses
bus	memory requests as seen on the bus

Table 2.3: An overview of the 34 execution characteristics we obtain from the hardware performance counters of the AMD Athlon XP. For each event, we measure the number of occurrences during the workload execution.

main memory and the number of writebacks. We also measure the L1 D-TLB misses that hit the L2 D-TLB and the L1 D-TLB misses that also miss the L2 D-TLB.

- **Bus unit**. We monitor the number of requests to main memory, as seen on the bus between the CPU and the main memory.

While we measure 34 events, we use only 33 characteristics. The latter are obtained by dividing the former by the number of retired instructions. By doing so, the characteristics are given per unit of execution. For example, one characteristic is the number of L1 D-cache misses per instruction. This performance measure is more appropriate than the usual L1 D-cache miss rate, because it is more related to actual performance. Indeed, even if the number of L1 D-cache misses per instruction is low, it can still result in a high L1 D-cache miss rate, if there are few L1 D-cache accesses.

All the events listed in Table 2.3 are measured for both user and kernel space execution, because Java applications execution spends a significant amount of time inside the kernel [69].

Remark 2.1. *The question can be raised why we do not normalise the events with respect to the number of elapsed cycles. We believe that dividing by the number of retired instructions is more suitable because the number of elapsed cycles during the execution is affected by several events, e.g., the number of data cache misses, stalls in the FPU, etc. The number of retired instructions on the other hand is fixed, if the execution is deterministic*[5].

2.4 Statistical analysis

We now elaborate on the statistical analysis techniques employed to deal with the large amount of data with the purpose of obtaining comprehensible results. Because there are 33 execution characteristics for 14 benchmarks, for 5 virtual machines, and for 2 input sizes, the total amount of data cannot be easily understood by a human observer. We therefore use statistical analysis, namely PCA and CA, for gaining insight in this large and complex data set. We use ANOVA for determining in a more qualitative manner whether the virtual machine or the benchmark contributes more to the observed variability. The following sections describe principal components analysis and cluster analysis in more detail. We also give a brief description of the purpose of ANOVA in this chapter.

[5]For a Java application this is not strictly true, see Chapter 3, but at least this number remains unperturbed by the other hardware events that occur.

2.4.1 Principal components analysis

Principal Components Analysis [62] (PCA) is a linear statistical technique that transforms an n-dimensional space into a space in which all dimensions are uncorrelated.

Given variables $X_i, i \in 1, \ldots, n$, PCA computes a set of n principal components

$$Z_i = \sum_j a_{ij} X_j \quad \text{for } i \in \{1, \ldots, n\} \tag{2.1}$$

such that

$$\text{Var}[Z_1] \geq \text{Var}[Z_2] \geq \ldots \geq \text{Var}[Z_n], \tag{2.2}$$

and

$$\sum_{i=1}^{n} \text{Var}[Z_i] = \sum_{i=1}^{n} \text{Var}[X_i].$$

As shown by Equation 2.2, some principal components will have a high variance, whereas others will have a small variance. This means that by dropping the principal components $Z_j, j \in p+1, \ldots, n$, we can retain a large portion of the variance (or information) that is present in the original data set, while simultaneously reducing the dimensionality of the space. This means that a human observer needs to deal with a less complex set of information. The retained fraction of variance is then given by

$$\frac{\sum_{i=1}^{p} \text{Var}[Z_i]}{\sum_{i=1}^{n} \text{Var}[X_i]}$$

Typically, one wants to retain p principal components that account for 75% to 90% of the original variance. Alternatively, one may want to retain principal components with a variance that is larger than unit, as these account for more variance than the original (normalised) variables.

In this study, the p original variables are the characteristics measured through the performance counters, normalised to a zero mean and unit standard deviation, prior to PCA. In this way, we attach equal weight to each characteristic. By examining the most important q principal components, which are linear combinations of the original performance events ($Z_i = \sum_{j=1}^{p} a_{ij} X_j, i = 1, \ldots, q$), meaningful interpretations can be given to these principal components in terms of the original execution characteristics. A coefficient a_{ij} that is close to $+1$ or -1 implies a strong impact of the original characteristic X_j on the principal component Z_i. A coefficient a_{ij} that is close to 0 on the other hand, implies little or no impact.

The next step in the analysis is to display the various Java workloads as points in the q-dimensional space spanned by the q principal components to view the Java workload space. Note again that the projection on the q-dimensional space will be easier to understand than a view on the original p-dimensional space for two reasons: (i) q is much smaller than p, $q \ll p$, and

(ii) the q-dimensional space is uncorrelated, making the Euclidean distance a feasible distance metric. On the other hand, the dimensions in the PCA-space are a combination of the original dimensions in the workload space, making them somewhat hard to interpret.

2.4.2 Cluster analysis

Cluster Analysis (CA) [62] is another data analysis technique that is aimed at finding a natural (or reasonable) clustering of the Java workloads into sets that exhibit similar behaviour. We will cluster data in the PCA space, instead of applying CA on the initial data. The motivation for this is that the original variables are highly correlated which implies that an Euclidean distance in this space is unreliable. First performing PCA alleviates this problem.

In this dissertation we use a hierarchical clustering strategy as well as a non-hierarchical strategy. There are two ways in which to obtain the former clustering: (i) divisive clustering, and (ii) agglomerative clustering. In the former approach, the initial single group of workloads is split up iteratively until the algorithm ends up with singleton sets, or sets of a certain minimum size. In the latter case – the approach used in this dissertation – the algorithm starts with the individual data points, i.e., there are as many sets as there are data points. Pairs of sets are joined iteratively to form larger sets until a single group containing all data points is obtained.

One particular agglomerative clustering technique is hierarchical linkage clustering. The algorithm will join sets based on the linkage distance between them, according to some metric. There are a number of metrics, but we will use the Euclidean distance, which is suitable, as explained above. Additionally, we rescaled the data prior to conducting CA. Each of the following linkage strategies cluster the two sets that are closest to each other. Hence, the strategies differ in the way they determine the distance between two sets. Essentially, there are three linkage strategies that are often used. Given two sets $A = \{a_1, \ldots, a_k\}$ and $B = \{b_1, \ldots, b_l\}$, the strategies can be described as follows. Denote the distance for a given distance metric d between workloads a and b as $|a - b|_d$.

- Single linkage: The distance between A and B is the minimum of

$$\min_{a \in A, b \in B} |a - b|_d$$

  . Thus, two groups are joined if the minimum distance between their members is the overall minimal distance between all group pairs. A major drawback of this approach is that it cannot make a distinction between poorly separated clusters [62].

- Complete linkage: The distance between A and B is the maximum of

$$\max_{a \in A, b \in B} |a - b|_d$$

. Consequently, two groups are joined if the maximum distance between their members is the overall minimum distance between all group pairs. This approach makes sure that all members of a cluster are within some maximal distance from each other.

- Average linkage: The distance between A and B is the average of $\{|a_i - b_j|_d \ | i \in \{1, \ldots, k\}, j \in \{1, \ldots, l\}\}$. Here, two groups are joined if the average distance between their members is the minimum distance between all group pairs. A example of average linking is the McQuitty linkage [79] strategy, which is used in this chapter.

The non-hierarchical clustering strategy we use is K-means. This iterative technique usually starts by choosing a random partitioning of all data points into a given number k of groups. In each iteration the data points x_i are reassigned to the cluster c_j with the centroid that is closest to it, i.e., the distance between x_i and c_j is minimal. Furthermore, the centroids are recomputed in each iteration. When no more data points are reassigned, the iteration ends.

Remark 2.2. *It is quite possible that two K-means clusterings applied to the same data result in a different clustering because the initial partitioning is chosen randomly. To address this issue, multiple clusterings should be tried. The final result will then be the clustering that shows the best result according to some (objective) score, e.g., the Bayesian Information Criterion [85].*

2.4.3 Analysis of variance

Analysis of Variance or ANOVA [62, 82] is technique that is aimed at explaining the relationship between exploratory variables or factors, e.g., the virtual machine or the benchmark, and the dependent variable – in this case the performance characteristics. We will give a more in-depth explanation of ANOVA in Chapter 4, but here it suffices to say that ANOVA splits up the total variance observed in the measurements into components that can be attributed to each factor, to the interaction between them, and into a component that accounts for the residual variance.

2.5 Results and discussion

In this evaluation section, we present and extensively discuss the results that were obtained from our analysis. First, we present the results for the smallest input sizes, i.e., *s1* for SPECjvm98 and *small* for DaCapo. Second, we look at how a larger input set – *s100* for SPECjvm98 and *large* for DaCapo – influences the conclusions we draw from the analysis. Third, we examine the complete workload space, i.e., the space in which we use all possible combinations of virtual machine, benchmark and input set.

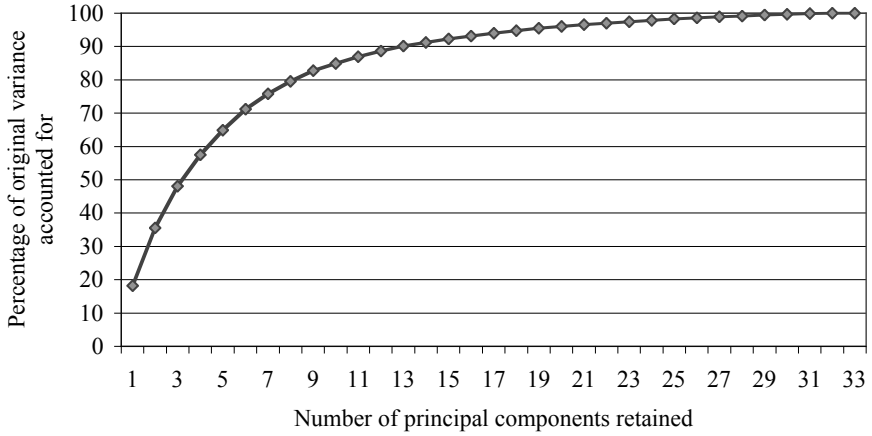


Figure 2.2: The percentage of variance, present in the original data, that is accounted for when retaining the first k principal components, for $k \in 1, \ldots, 33$.

2.5.1 Workloads with small input sets

We first present a detailed analysis for the SPECjvm98 benchmarks using the *s1* input set. We then briefly discuss the most relevant items for the DaCapo benchmarks with the *small* input set.

PCA factor loadings for SPECjvm98

For SPECjvm98 with the *s1* input set, PCA yields the following results and insights. First of all, to account for a reasonable amount of variance present in the original data, we need to retain at least seven (accounts for 79.60% of total variance) or eight (accounts for 82.73% of total variance) principal components (PCs), see also Figure 2.2. The factor loadings are given in Figures 2.3 and 2.4. The former figure shows the loadings for the first four principal components, the latter shows the loadings for the last four principal components. They account for 18.22%, 17.35%, 12.53%, 9.36%, 7.45%, 6.28%, 4.59%, 3.83% of the total variance, respectively. The first and second components are more or less equally important. The contributions of the other components are relatively smaller. In the following enumeration we discuss the most important contributions made by the execution characteristics to each of the first eight PCs.

- The main positive influences on the first PC is given by far control transfers, stalls (dispatch, full ICU, full LSU to both L1 and L2 cache), data cache misses and requests to main memory. Negative influences are taken branches, near returns, fetches to the instruction cache, branch

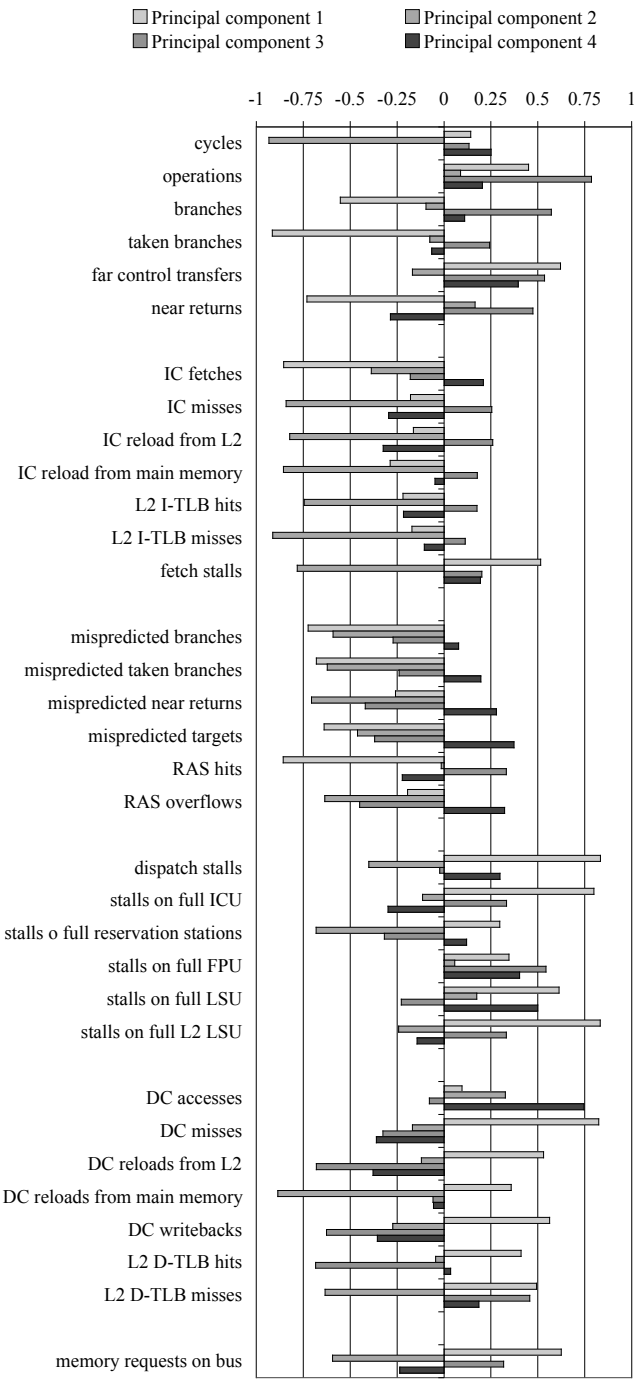


Figure 2.3: The factor loadings for the first four principal components for the SPECjvm98 benchmarks with the *s1* input set.

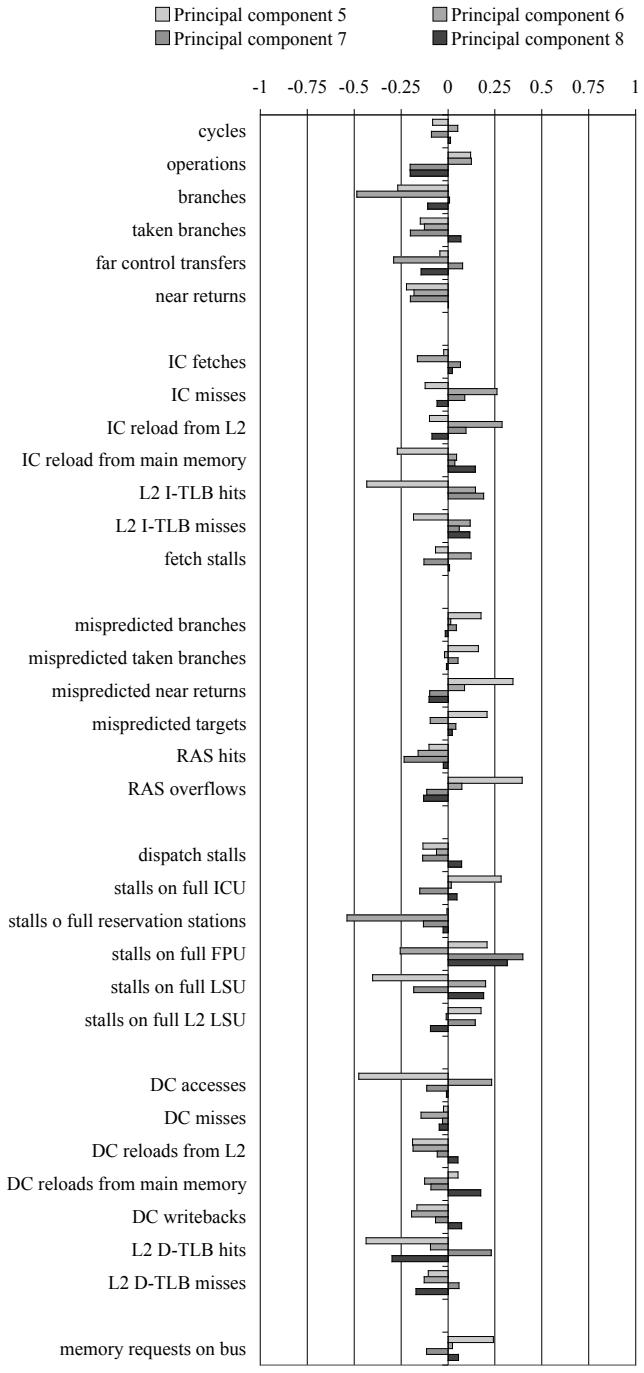


Figure 2.4: The factor loadings for the second four principal components for SPECjvm98 benchmarks with the *s1* input set.

prediction (mispredictions for branches (taken or not), mispredicted targets and hits in the return address stack (RAS)).

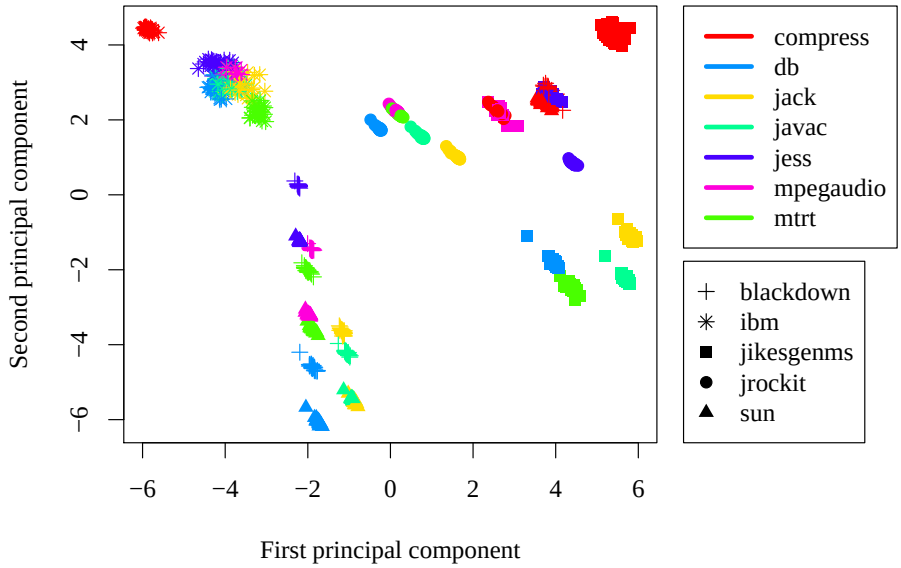
- The main positive contribution to the second PC is made by the data cache accesses (0.33). There are, however plenty of negative contributions, such as the cycles, practically every event from the processor front-end, branch mispredictions events and RAS overflows, full reservation stations, data cache reloads from main memory, and L2 data TLB misses.
- The third PC has a single large positive contribution, i.e., the number of macro-operations, and two minor contributions from full FPU events (0.54) and far control transfers (0.53). Negative contributions are made by data cache refills from the L2 cache, data cache writebacks and L2 data TLB hits.
- The fourth PC obtains a single large positive contribution from the data cache access events. The largest negative contribution is made by the data cache refills from the level 2 cache (-0.38).
- Although the fifth through eight principal components still contribute significantly to the percentage of total variance accounted for, there are but a few characteristics that contribute to each of these PCs. For example, the fifth PC gains positive contributions from mispredicted near returns and RAS overflows. Negative contributions are made by full LSU stall events, data cache accesses and writebacks from the data cache to main memory. The sixth PC has but a single large negative contribution from the full reservation station events. The last two retained PCs take a large positive contribution from full floating point unit stalls.

The factor loadings also give an indication of the correlated characteristics for this set of Java workloads. For example, from these results we can conclude that (along the first principal component) the branch characteristics correlate well with the front-end characteristics. Moreover, this correlation is a positive correlation since both characteristics have a positive contribution to the first principal component. Also, the front-end characteristics correlate negatively with the amount of fetch stalls. In other words, this implies for example that a high number of I-cache fetches per unit of time correlates well with a low number of fetch stalls per unit of time, which can be understood intuitively.

PCA space for SPECjvm98

Now that we know the loadings, we can place the workloads in the PCA space. For convenience, we only show the space spanned by the first four principal components, see Figure 2.5. In the top graph, we project the PCA space onto the subspace spanned by the first two principal components; in the bottom graph we project the PCA space onto the subspace spanned by the third and fourth principal. In these figures, the virtual machines are identified

(a) First and second principal component



(b) Third and fourth principal component

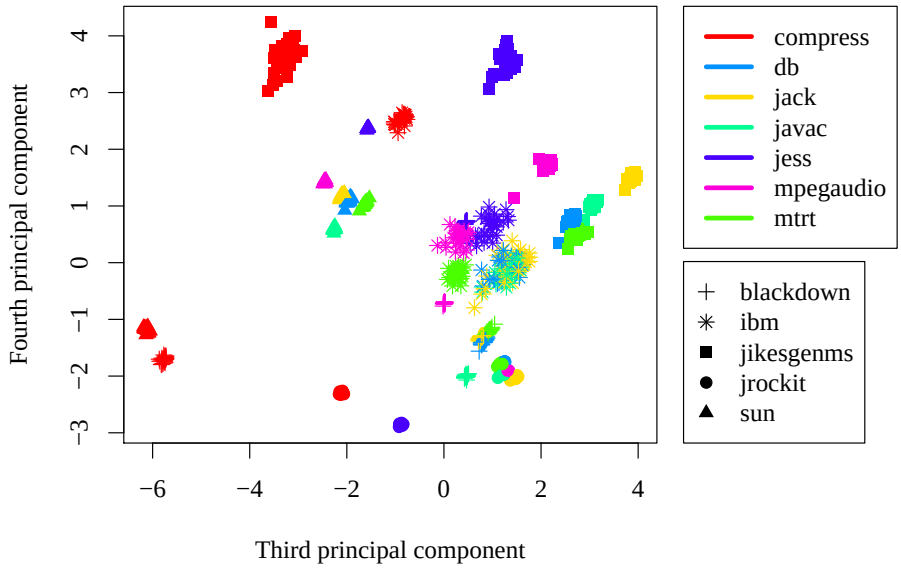
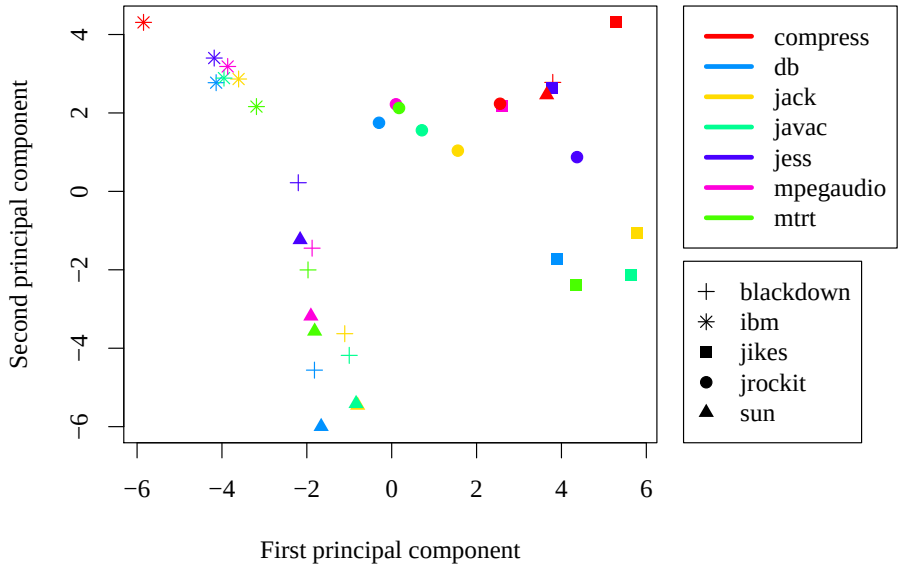


Figure 2.5: Scatter plots for SPECjvm98 with the *s1* input set. In the PCA, all 30 measurements for each workload were used. They are shown as individual points in the graphs.

(a) First and second principal component



(b) Third and fourth principal component

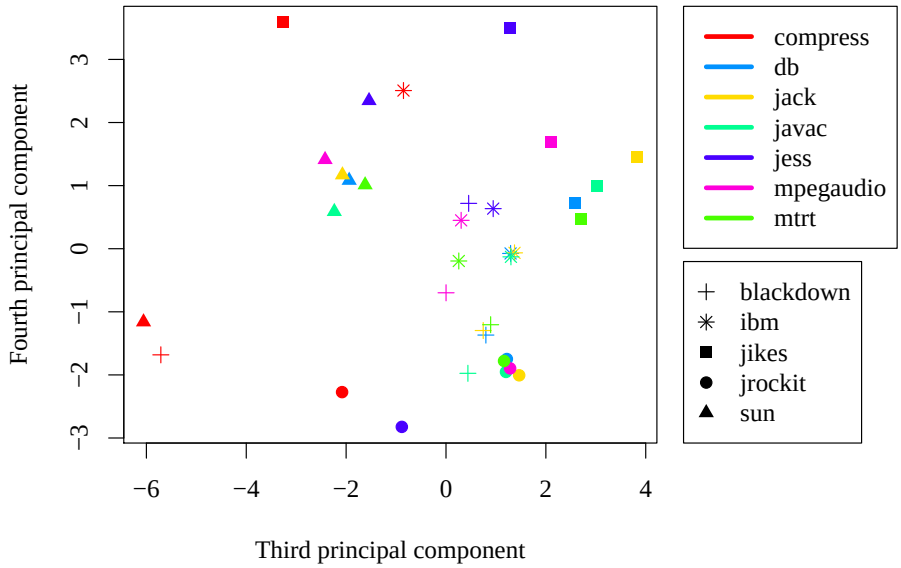


Figure 2.6: Scatter plots for SPECjvm98 with the *s1* input set. In the PCA, we use the average value from 30 measurements for each workload.

by the symbol used, whereas the benchmarks are identified by colours. Since we are dealing with a four-dimensional space, it is important to consider these two plots simultaneously to get a clear picture of the four dimensions.

Remark 2.3. *The location of a workload in the PCA space is determined by both the factor loadings and the relative values of the execution characteristics. The former merely provide the weight factors a_{ij} in the PCA Equation 2.1. To understand the meaning of the positioning in the PCA space, it is important to keep the following in mind. Because the characteristics are normalised to have a zero mean and a unit standard deviation, positive scores for a PC generally mean that the workloads have an above average value for the characteristics with positive loadings and a below average score for characteristics with a negative loading.*

Remark 2.4. *The most important information we can derive from the PCA is the relative position of the workloads in the PCA space. But even workloads that lie close together need not have highly resembling execution characteristics. Indeed, given two workloads W_a and W_b, the scores pc_a and pc_b for a given component might be more or less the same, even if the contributing characteristics are not. Essentially, if we limit the number of dimensions retained, PCA amounts to a surjection from the workload space to the PCA space. Still, in practice, similar PCs generally imply similar original characteristics.*

It is quite clear from the graphs in Figure 2.5 that the workloads are clustered per virtual machine, rather than per benchmark. This is even more visible in Figure 2.6, which shows the same graphs, but where average values are used during PCA. Especially, the workloads with the IBM J9 virtual machine are clustered tightly together. In the PCA space made up by the first two PCs, J9 is located on the top left of the graph. In the bottom figure, the J9 workloads (except for *compress*) still form a tight cluster, but they have some overlap with other virtual machine clusters. In the PC1-PC2 plot, we also observe a tight intertwinement of the Blackdown and SUN virtual machines. This is not the case in the PC3-PC4 space, where both virtual machines seem to form a separate cluster. Still, it can be expected their behaviour is quite similar, because they both are based on the HotSpot VM technology. However, in our experiments, SUN is a virtual machine for Java 1.5, whereas Blackdown is virtual machine for Java 1.4.1. In the PC1-PC2 space, Jikes RVM form either a large cluster on the right, or two separate clusters, one with three benchmarks, the other with four benchmarks. Figure 2.7 illustrates a particular argument why the Jikes workloads are lying far from the IBM J9 workloads in the first PC dimension. Apparently, Jikes has (relatively) much more L1 data cache misses compared to the IBM virtual machine. In the PC3-PC4 space, only *compress* is located apart from the Jikes RVM cluster. Finally, JRockit has some overlap with Jikes RVM in the top figure, and some overlap with the Blackdown cluster in the bottom figure.

The only benchmark for which the data points are consistently lying apart from the virtual machine clusters is *compress*. To put it differently, the interaction between *compress* and the virtual machine it runs on, has a large impact

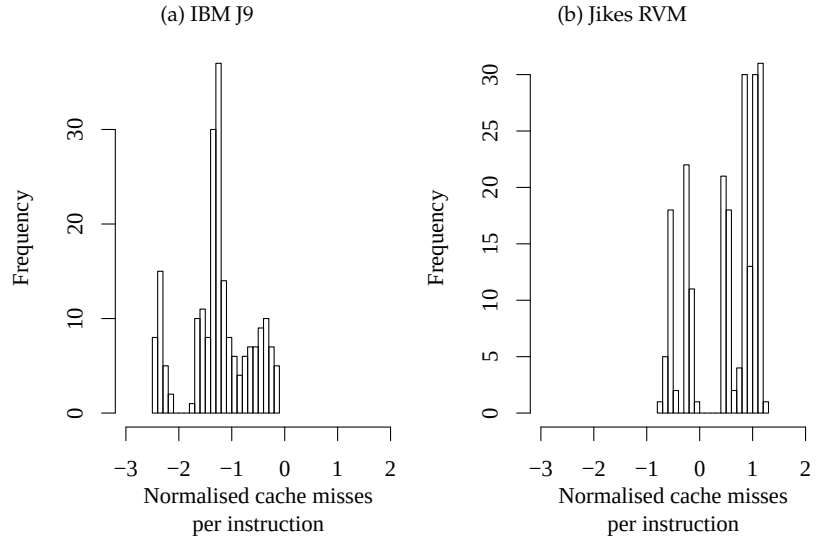


Figure 2.7: Comparison between the normalised number of L1 data cache misses per instruction for the IBM J9 virtual machine (left) and the Jikes RVM (right).

on its overall behaviour at the microarchitectural level or the execution characteristics. The reason for this phenomenon is that *compress* has a very small code size, with basically 10 methods that account for 98% of all method calls. On the other hand, *compress* is processing a fairly large amount of data, even in the case of the *s1* input set. The small code size allows for aggressive optimisation by the virtual machines' compilers, and different virtual machines lead to different optimised code versions.

PCA space for the DaCapo benchmarks using the *small* input set

Figure 2.8 shows similar results for the DaCapo suite. While the influence of the benchmarks themselves seems a bit more pronounced, we can still distinguish clusters per virtual machine. For these workloads, the first eight principal components account for 74.91% of the total variance (17.95%, 16.01%, 9.59%, 8.57%, 7.57%, 6.71%, 4.39%, 4.11%, respectively). Note though that the factor loadings for these benchmarks differ from those for SPECjvm98 that are shown in Figures 2.3 and 2.4.

Once again, a tight cluster can be observed for the IBM J9 workloads (with the notable exception of *jython*). In the space spanned by the first two principal components, J9 resides more or less on the left side for PC1, and central for PC2. The latter means that large contributions to PC2 neutralise each other (or that there are no large contributions). Jikes RVM has a small overlap with JRockit in the top figure, and a larger overlap with SUN in the bottom figure. The latter overlaps with Blackdown in the PC1-PC2 space, but not in the PC3-

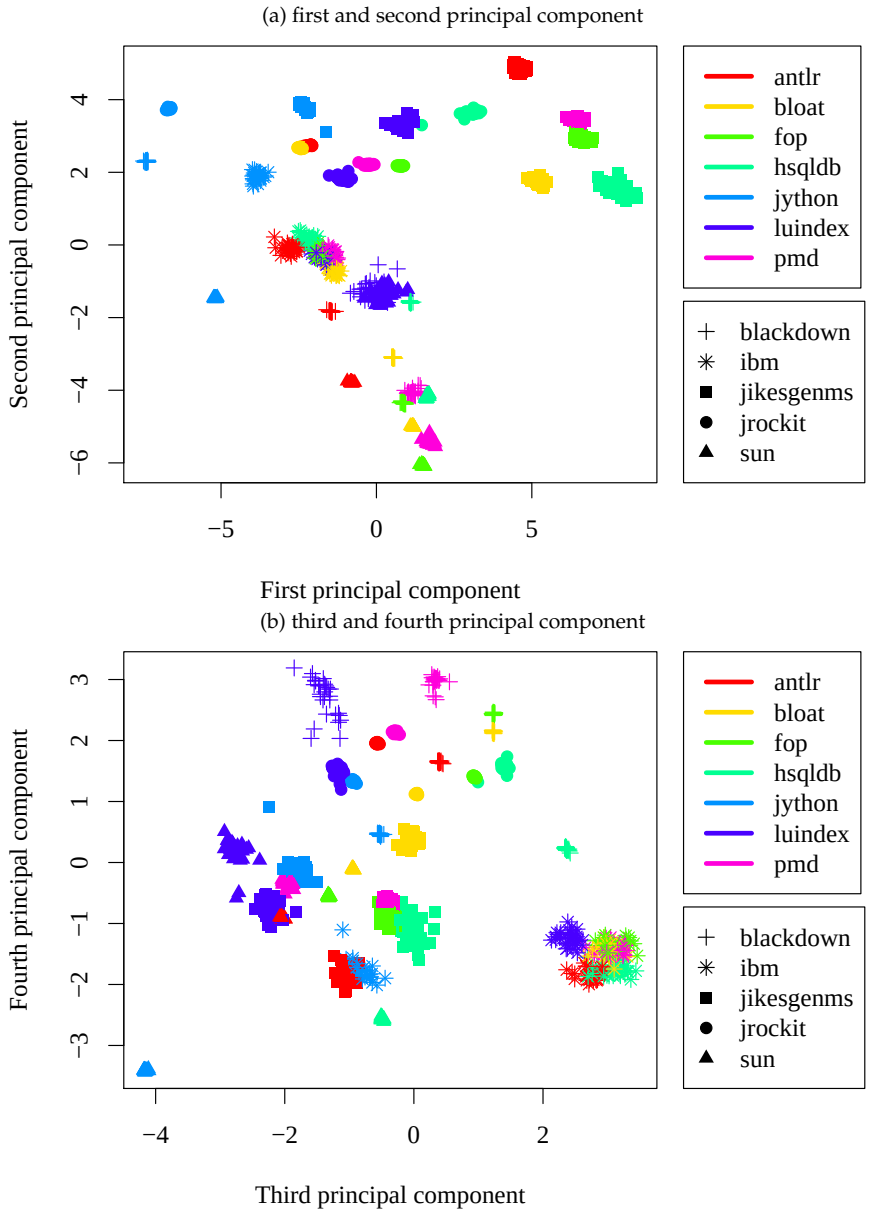


Figure 2.8: Scatter plots for DaCapo with the *small* input set. In the PCA, all 30 measurements for each workload were used.

PC4 space.

In Figure 2.8, we can also see that the benchmarks make a larger contribution to the overall behaviour than is the case for SPECjvm98 benchmarks with small input sets. For example, in the top graph, the clusters formed by both *jython* and *luindex* are no larger than any of the virtual machines clusters we identified. The other benchmarks remain more spread in the space. The effect is even more pronounced in the bottom graph. The benchmarks *bloat*, *jython*, and *luindex* can be identified as belonging to a cluster comprised of different virtual machines; however, no IBM J9 workload is present in these clusters.

Remark 2.5. *The purpose of including small input sets for the benchmarks in the DaCapo suite is to provide input sets that allow testing the application and the virtual machine. For experimental evaluation, the DaCapo group recommends using either the default or large input set.*

From these graphs we conclude that for the small input sets (*s1* for SPECjvm98 and *small* for DaCapo), the virtual machine has a larger impact on the overall behaviour than the Java application that is being executed. Consequently, a virtual machine running a Java application with a small input set will likely exhibit similar behaviour irrespective of the particular application and input set. This can be understood intuitively because the *s1* input set results in very short benchmarks for which the virtual machine start-up (initialising and loading significant parts of the Java library, compiling methods) makes the more important contribution to the overall behaviour. This shows the pitfall under discussion is quite tangible.

CA for SPECjvm98 and DaCapo

Table 2.4 shows the result obtained by conducting a K-means clustering targeting the formation of 10 clusters[6]. The reinforces our earlier finding: the virtual machines largely determine the clusters. For two clusters (2 and 4) there are two virtual machines in the clusters, on both occasions for a single benchmark; *mpegaudio* in cluster 2 and *compress* in cluster 4.

We can cluster the data in a hierarchical manner as well. For such a clustering, a dendrogram provides an elegant way to visualise the resulting clusters, see Figures 2.9 and 2.10. Simply put, a dendrogram is a more fancy form of a binary tree. As we mentioned earlier, during the clustering process, two clusters are linked together when they lie closest together according to some distance metric. Thus, a dendrogram graphically shows the linkage distance at which clusters are joined. The idea is that workloads clustered together early on – and hence lie next or close to each other in the dendrogram – exhibit similar behaviour as measured by the performance characteristics. Workloads that are clustered late in the iterative process lie further apart and tend to show dissimilar behaviour. To identify actual clusters in the dendrogram, it is easiest

[6]Note that this is an arbitrary number.

Cluster	Virtual machine	Benchmark
1	Jikes	mpegaudio
	Jikes	mtrt
2	Blackdown	mpegaudio
	SUN	mpegaudio
3	SUN	jess
	SUN	db
	SUN	javac
	SUN	mtrt
	SUN	jack
4	Blackdown	compress
	SUN	compress
5	JRockit	jess
	JRockit	db
	JRockit	javac
	JRockit	mtrt
	JRockit	jack
6	Blackdown	jess
	Blackdown	db
	Blackdown	javac
	Blackdown	mtrt
	Blackdown	jack
7	Jikes	jess
	Jikes	db
	Jikes	javac
	Jikes	jack
8	Jikes	compress
9	IBM	compress
	IBM	jess
	IBM	db
	IBM	javac
	IBM	mpegaudio
	IBM	mtrt
	IBM	jack
10	JRockit	compress
	JRockit	mpegaudio

Table 2.4: Resulting clustering of a K-means clustering on the SPECjvm98 benchmark data with the *s1* input set, where 10 clusters were withheld.

to cut the tree at a certain linkage distance, see for example Figure 2.9 where the blue line indicates the cut. The sub-trees that dangle from the cutting position are then considered to form a single cluster. We have opted to cut the dendrogram such that 10 clusters are marked, the same number as we use for the K-means clustering.

From the dendrogram in Figure 2.9, we can make the following observations. The top cluster is comprised of workloads running the SUN virtual machine, except for the first workload – *mpegaudio* on Blackdown – which is linked to the rest of the cluster at a distance almost approaching the linkage distance of the cut. The next interesting cluster is made up of two sub-clusters: one with Jikes RVM workloads, the other with *compress* on the SUN and Blackdown virtual machines. The fourth cluster contains but IBM J9 workloads, while the fifth is made up of JRockit and Jikes RVM workloads. The other clusters are relatively small, and mostly contain mixes of virtual machines.

The dendrogram in Figure 2.10 has three singleton clusters at the chosen linkage distance. We further observe (top to bottom) the following clusters. The second cluster is comprised of Blackdown workloads, except for *jython* on JRockit, which lies nearest to *fop* on Blackdown. The next clusters contain both Jikes RVM and JRockit workloads, with a single Blackdown workload (*luindex*) thrown in. We also observe a cluster with solely SUN workloads, as well as a larger cluster with only IBM J9 workloads. Clearly, for the DaCapo benchmarks, the influence of the virtual machine is not as dominant as was the case for SPECjvm98.

Generally speaking, we conclude that *for short running workloads, the virtual machine brings about the largest impact on the overall behaviour on the first iteration of the benchmark*. In the PCA space, the benchmarks are mostly grouped together by virtual machine, rather than by benchmark. This is reflected in the cluster analysis we conducted in this space.

Variability for DaCapo using the small input set

Figure 2.11 shows a breakdown of the total observed variability for each of the 33 characteristics. The graph shows the percentage of the variability accounted for by the virtual machine, the benchmark, the interaction between virtual machine and benchmark as well as the residual variability. It is clear that the virtual machine accounts for the largest percentage of the overall variability. Yet, an ANOVA indicates that for each of the characteristics the virtual machine, benchmark and their interaction have a significant effect at the 95% confidence level. Remarkably, for the CPI, the data cache misses, the refills from both the L2 cache and the system, the application takes the upper hand.

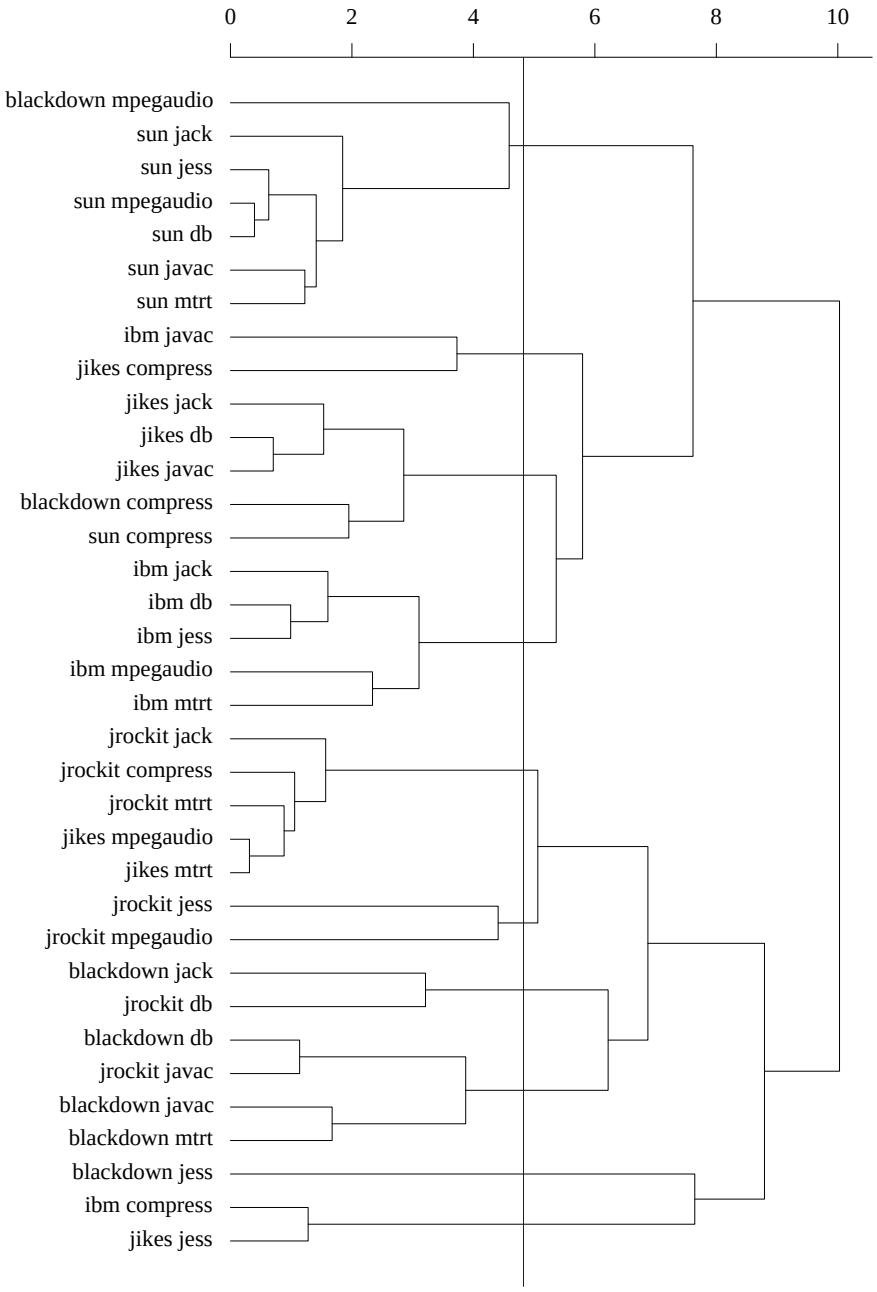


Figure 2.9: Dendrogram representing the hierarchical clustering of the SPECjvm98 benchmarks with the *s1* input set using a McQuitty average linkage clustering algorithm.

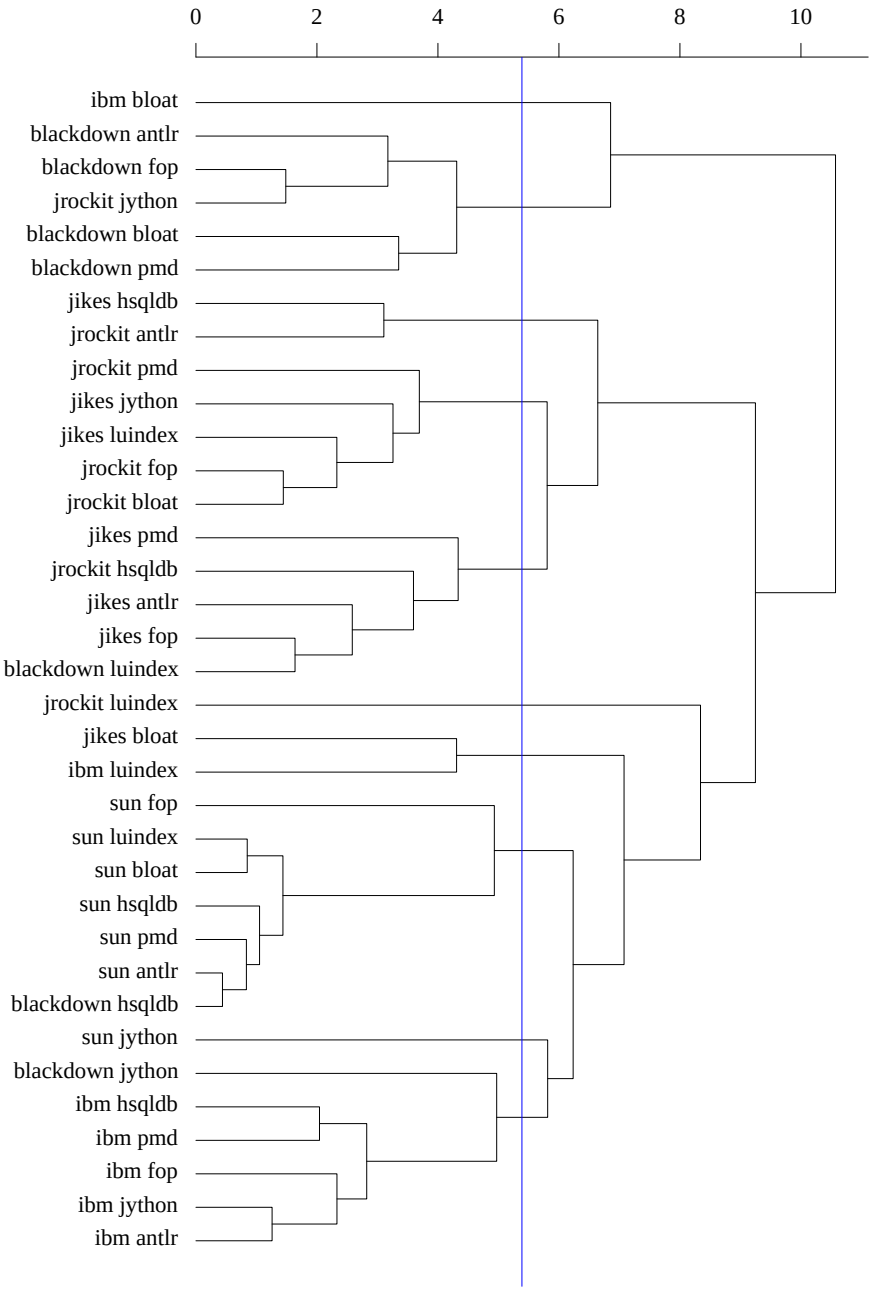


Figure 2.10: Dendrogram representing the hierarchical clustering of the DaCapo benchmarks with the *small* input set using the McQuitty average linkage clustering algorithm.

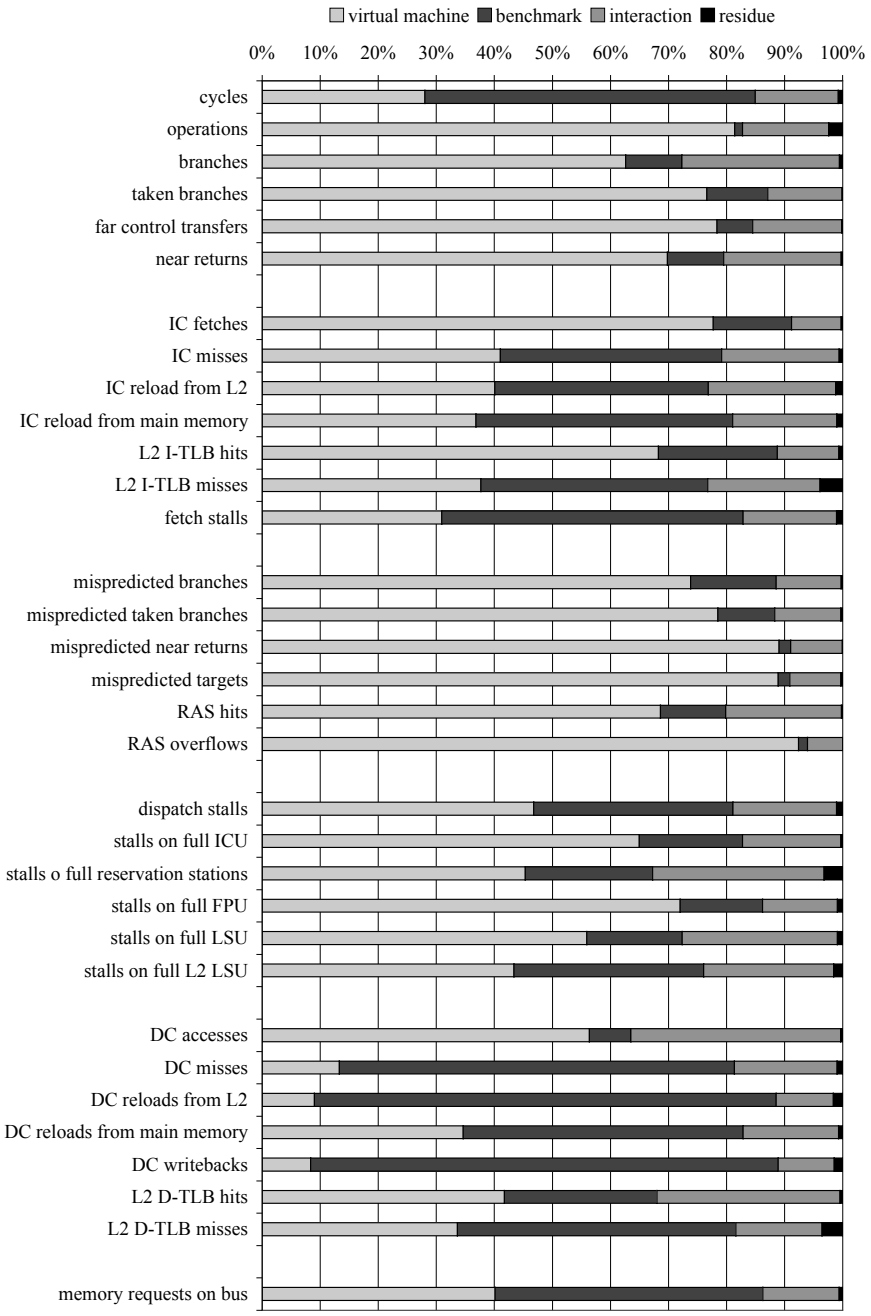


Figure 2.11: Breakdown of the variability for each characteristic for DaCapo with the *small* input set accounted for by (i) the virtual machine, (ii) the benchmark, (iii) their interaction, and (iv) the residual variability.

2.5.2 Workloads with large input sets

We now consider Java workloads with large input sets, i.e., these inputs cause the Java applications to run for a longer period of time.

For the SPECjvm98 benchmarks with the *s100* input set, we need to retain at least eight (accounts for 77.69%) principal components. These components account for 20.76%, 17.17%, 9.30%, 8.31%, 6.56%, 5.72%, 5.19%, and 4.67% of the total variance, respectively. The first and second components seem almost equally important. From the third component onward, the additionally explained variance drops off.

The execution characteristics make the following contributions to the principal components:

- The first PC is positively influenced, mainly by the cycles[7], fetch stalls and other non-FPU related stalls, data cache events (except accesses) and request to main memory. There are no particular large negative influences; the processor front-end events provide small negative contributions.
- The main positive contributions to the second PC are made by the processor front-end characteristics, the mispredicted branches and taken branches, and the number of near returns. The single negative contribution worth mentioning is the stall event when the FPU is full.
- The third PC only has one small positive contribution made by the number of mispredicted near returns (0.44). On the other hand, negative contributions are made by the number of branches and taken branches.
- From the fourth PC onward, there are almost no contributions worth mentioning. Exceptions are the full FPU events (-0.58) and the hits in the level 2 data TLB (0.59) that contribute to the fifth PC, as well as the number of data cache accesses (0.68) that contribute to the sixth PC .

Figures 2.12 and 2.13 show the first four dimensions of the PCA space for SPECjvm98 *s100* and DaCapo *large*, respectively. It is striking to observe that the workloads are now much more grouped by benchmark (i.e., by colour), especially on the graphs depicting the first two dimensions of the PCA space. In the higher dimensions this is less pronounced, as we can easily identify both benchmark and virtual machine clusters.

A K-means clustering confirms the observations we made above. Table 2.5 shows the resulting clustering for DaCapo with a *large* input set, obtained from 100 K-means trials for which the clustering with the optimal BIC score was withheld. In fact, it seems to indicate that the virtual machine still holds some sway over the overall behaviour. We find there are several clusters containing only workloads for a single virtual machine. Others contain but a

[7] The actual performance characteristic is thus CPI, or cycles per instruction

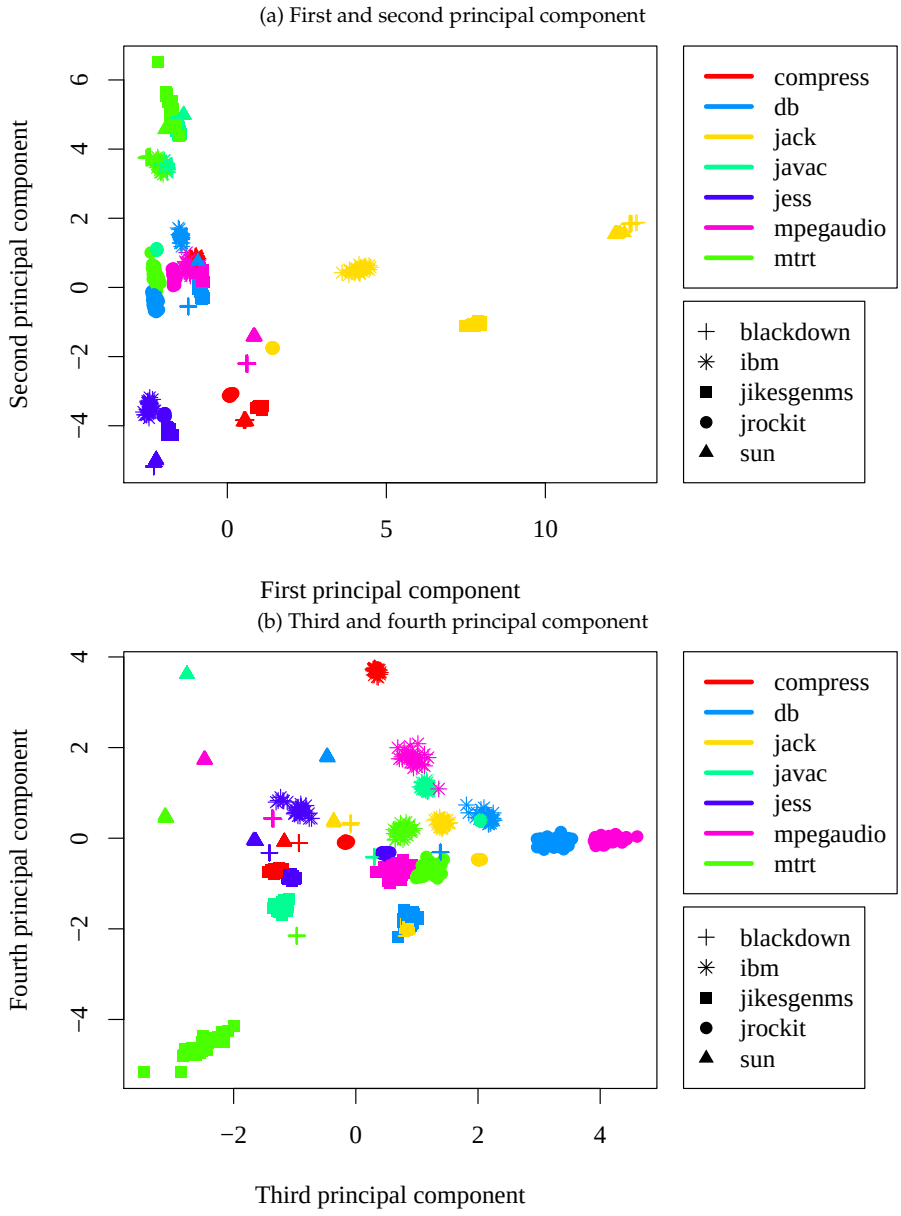


Figure 2.12: Scatter plots for SPECjvm98 with the s100 input set. In the PCA, all 30 measurements for each workload were used.

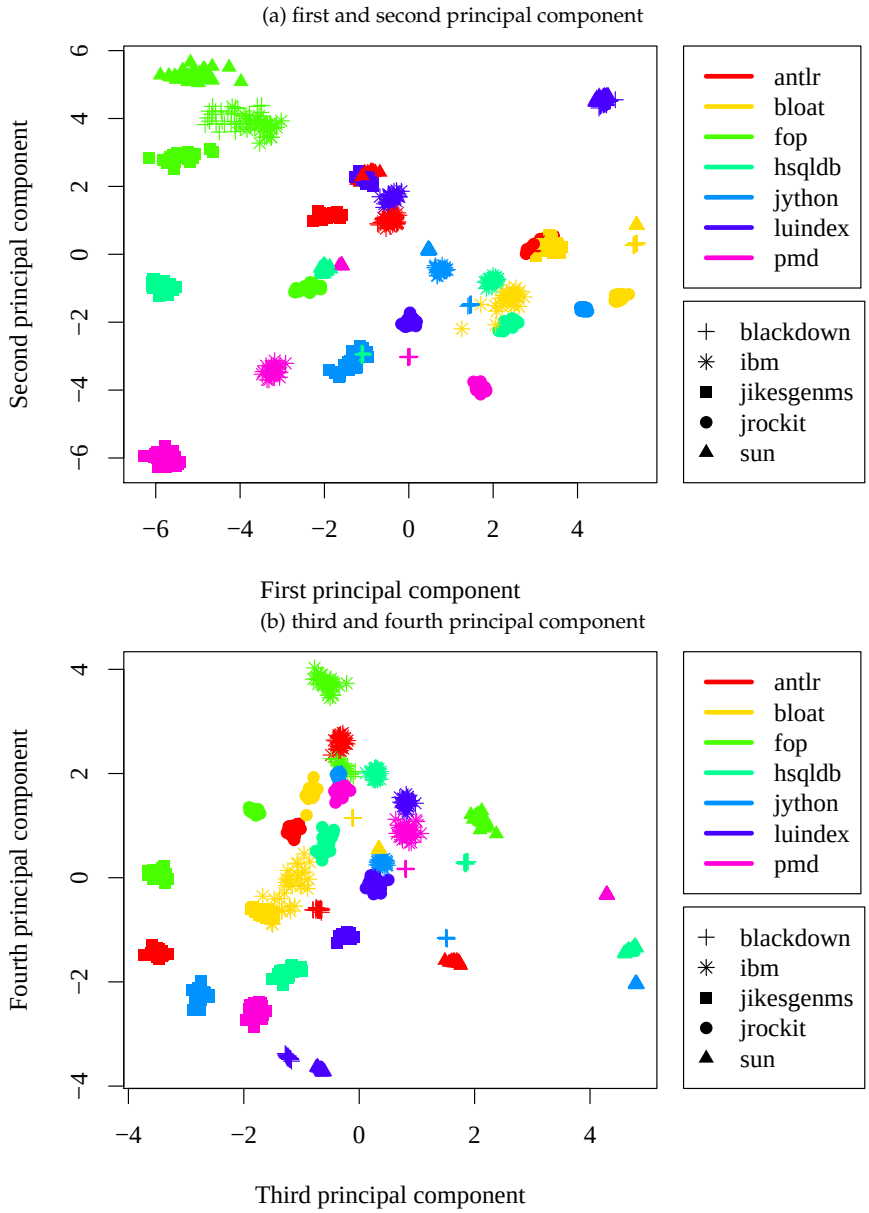


Figure 2.13: Scatter plots for DaCapo with the large input set. In the PCA, all 30 measurements for each workload were used.

Cluster	Virtual machine	Benchmark
1	IBM	bloat
	IBM	hsqldb
	JRockit	antlr
	JRockit	hsqldb
	JRockit	jython
	JRockit	pmd
2	Blackdown	hsqldb
	IBM	pmd
	JRockit	fop
	JRockit	luindex
3	Jikes	antlr
	Jikes	fop
	Jikes	hsqldb
4	SUN	antlr
	SUN	hsqldb
	SUN	jython
	SUN	pmd
5	Blackdown	jython
	Blackdown	pmd
6	Blackdown	luindex
	SUN	luindex
7	Blackdown	antlr
	IBM	antlr
	IBM	jython
	IBM	luindex
	Jikes	luindex
8	Blackdown	fop
	IBM	fop
	SUN	fop
9	Blackdown	bloat
	Jikes	bloat
	JRockit	bloat
	SUN	bloat
10	Jikes	jython
	Jikes	pmd

Table 2.5: Result of a K-means clustering on the DaCapo benchmark data with the *large* input set, building 10 clusters.

single benchmark, e.g., cluster 8 with *fop* and cluster 9 with *bloat*. But even for these clusters there is at least one workload for the same benchmark that got clustered elsewhere. We also see clusters containing both multiple virtual machines and multiple benchmarks, e.g., clusters 2 and 7.

The dendrograms for both SPECjvm98 and DaCapo point towards the same conclusion as the K-means clustering did. Figure 2.14 – showing DaCapo benchmarks with the *large* input set – has one very large cluster, which is well mixed in terms of virtual machines and benchmarks. There are two singleton clusters, and there is one cluster with only JRockit workloads (*pmd*, *antlr* and *jython*).

Figure 2.15 shows the breakdown of the variability, similarly to Figure 2.11, yet now for DaCapo with the *large* input set. Here, the application plays a more prominent part for almost all of the characteristics, yet the VM still has a significant impact on the execution.

We can conclude that using *longer running benchmarks causes the application to exert more influence on the overall behaviour, but still the virtual machine plays an important role*. This means, that even for long running benchmarks, researchers should take this into account and do their experiments, if possible, using multiple virtual machines and multiple benchmark iterations. Using multiple benchmark iterations is motivated by the fact that in a real application server environment, programs are often run more than once inside a single virtual machine invocation.

2.5.3 Small versus large input sets

One of the questions we want to answer is whether small input sets are representative for large input sets, e.g., are the performance numbers we obtain for *db s1* indicative of the performance numbers for *db s100*? In Figure 2.16, we show the clustering of the SPECjvm98 benchmarks with both input sets after the data has been transformed through PCA (we retain 5 principal components, accounting for over 80% of the variance) on both a SUN virtual machine and the Jikes RVM. We are not considering multiple virtual machines in a single analysis for the following reason. In the previous sections, we saw that the virtual machine has a lot of impact on the overall behaviour, and this additional influence is undesirable in this experiment.

Should the smaller input set size yield a representative execution for a larger size, the workloads would be clusters per benchmark, given the fact that we only consider a single virtual machine in each dendrogram. This, however, is not the case. In the dendrogram for the SUN virtual machine, see Figure 2.16, *jack s1* is linked with the cluster containing *jack s100* at the second highest linkage distance. Similarly, *db s100* and *db s1* are in separate clusters, linked at the highest linkage distance. Figure 2.17 confirms these findings for DaCapo, where only the *hsqldb* workloads get linked to each other immediately on the Blackdown virtual machine. Therefore, we conclude that *one*

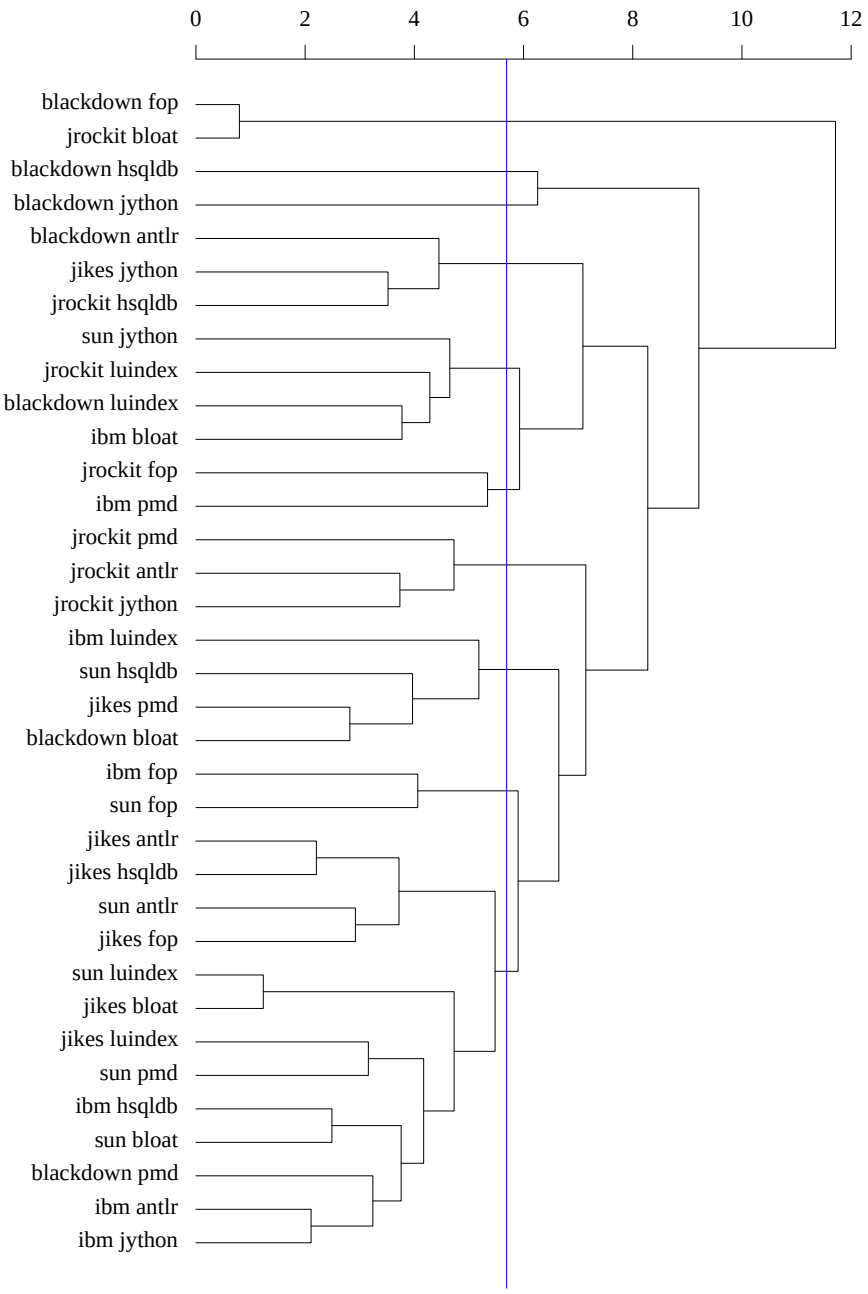


Figure 2.14: Dendrogram representing the hierarchical clustering of the DaCapo benchmarks with the *large* input set using the McQuitty average linkage clustering algorithm.

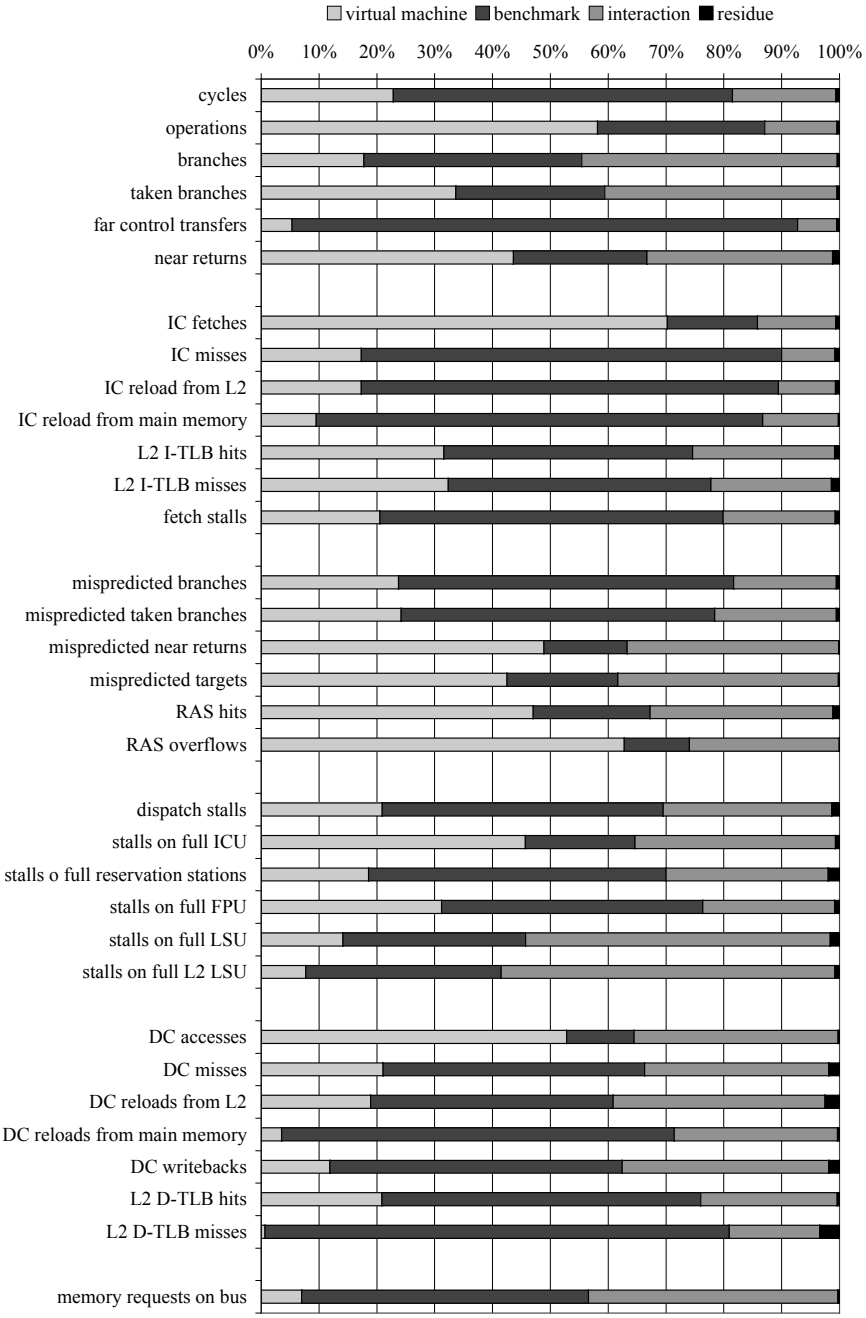


Figure 2.15: Breakdown of the variability for each characteristic for DaCapo with the *large* input set accounted for by (i) the virtual machine, (ii) the benchmark, (iii) their interaction, and (iv) the residual variability.

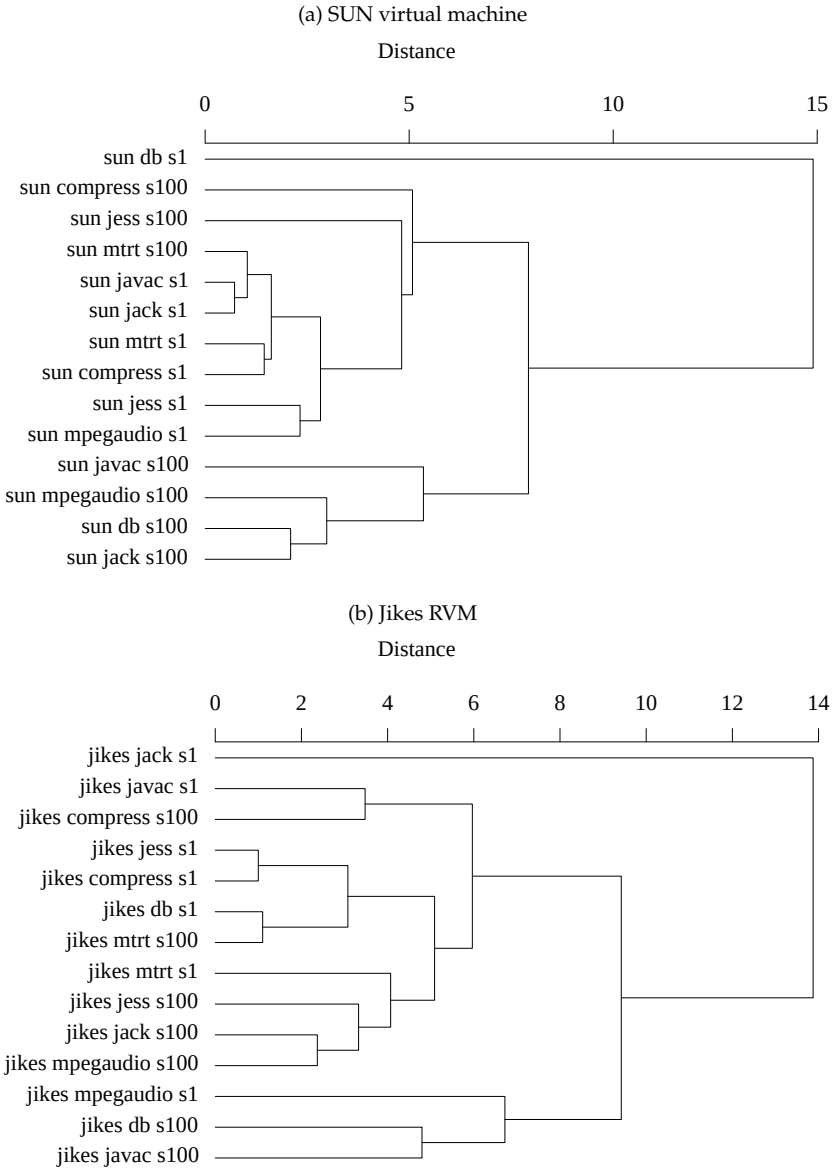


Figure 2.16: Dendrogram showing the clustering of the SPECjvm98 benchmarks with both the *s1* and the *s100* input sets for (a) the SUN virtual machine, and (b) Jikes RVM.

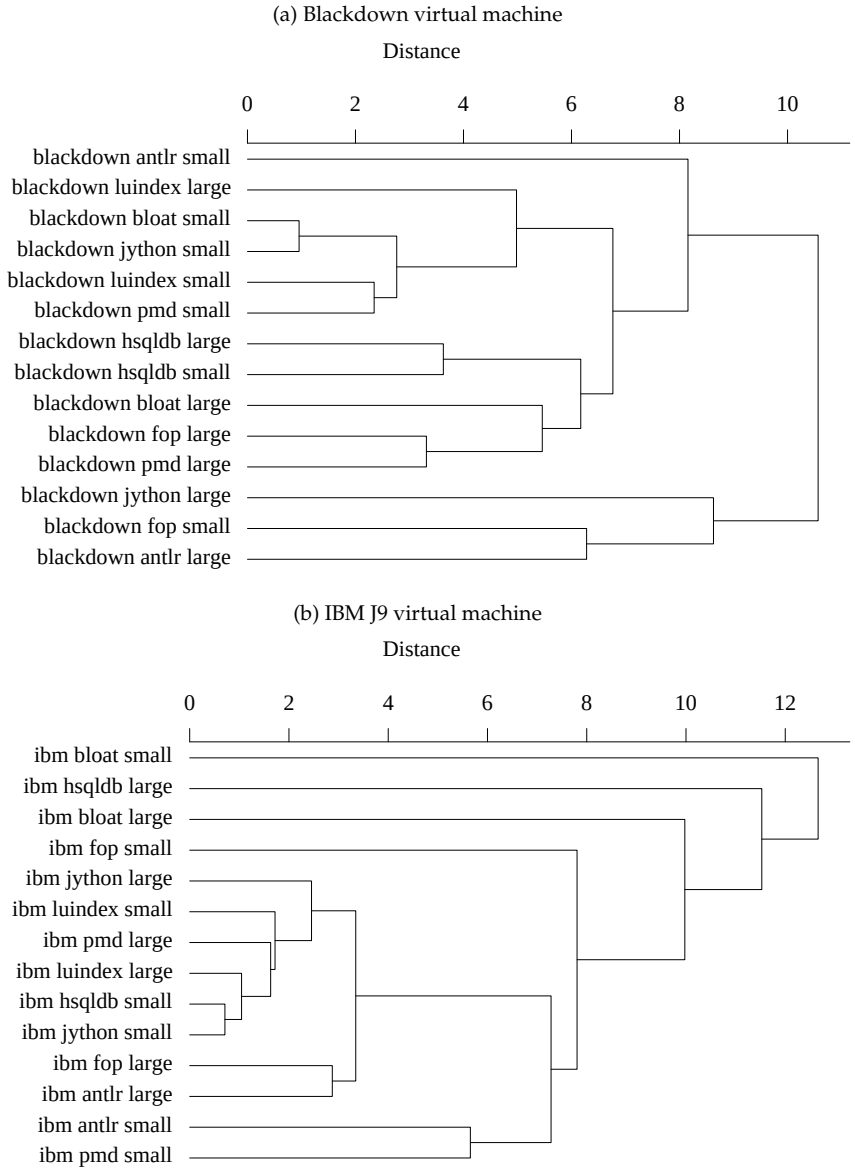


Figure 2.17: Dendrogram showing the clustering of the DaCapo benchmarks with both the *small* and the *large* input sets for (a) the Blackdown virtual machine, and (b) the IBM J9 virtual machine.

should not use a small input set size to infer results for a large input set size.

2.5.4 All the Java workloads

We now consider all benchmarks, virtual machines and input sets in one analysis, and discuss its implications for virtual machine developers, software developers, and computer architects. Based on the principal components analysis we retain 10 principal components from the PCA space, where we perform a K-means clustering. We set up the latter analysis to determine 40 clusters, as shown in Tables 2.6 and 2.7. We can make several interesting observations from this clustering.

- First, we can observe a number of virtual machine clusters for workloads with a small input set, notably clusters 1, 16, 17, 18, 20, 23, 28, and 38. Although in each of these clusters there are several benchmarks present, it is clear that for these workloads, the virtual machine holds sway over the behaviour.
- In the clusters where but a single benchmark occurs, the input size is either *s100* or *large*. Examples are the clusters 4, 5, 15, 22, 25, 27, and 40. In but a single cluster the input size was *small*: cluster 37 with the *luindex* benchmark running on the SUN and Blackdown virtual machines. The cluster for *compress* contains both *s1* and *s100* input sizes.
- There are several clusters in which a single virtual machine deals the cards, for various combinations of both benchmark and input set: clusters 3, 6, 9, 11 (with the exception of the single Blackdown workload), 21 (where a single IBM J9 workload resides as well), 30, and 39.

The above K-means clustering also provides an opportunity to select a (small) number of workloads such that a large portion of the PCA-space, and thus the workload space is well covered, e.g., for speeding up initial simulation-based architectural studies. How these results should be interpreted and used by researchers in the object-oriented programming community depends on their research goals. Because the virtual machine has a significant impact on overall behaviour, virtual machine developers benchmarking should select a number of benchmarks that cover a sufficiently large behavioural spectrum for their virtual machine. The collection of benchmarks will thus be different for different virtual machines. For example, for JRockit we recommend *mpegaudio*, *luindex*, *javac*, *bloat*, *fop*, and *mtrt*. We choose these benchmarks by picking a single workload from the clusters containing JRockit. Java application developers benchmarking their own Java program are recommended to use a sufficiently large number of virtual machines.

Of course, benchmark subsetting can be dangerous, because it can be used to game the results [86]. This means that, e.g., during architectural exploration, the design choices can be influenced for the worse or the better by

Cluster	VM	Benchmark	Input
1	Jikes	bloat	small
	Jikes	fop	small
	Jikes	hsqldb	small
	Jikes	pmd	small
2	JRockit	mpegaudio	s100
3	Jikes	javac	s100
	Jikes	jack	s100
	Jikes	fop	large
	Jikes	hsqldb	large
4	Blackdown	mtrt	s100
	SUN	mtrt	s100
5	Blackdown	db	s100
	SUN	db	s100
6	IBM	javac	s100
	IBM	jack	s100
	IBM	antlr	large
	IBM	antlr	small
	IBM	bloat	small
	IBM	fop	small
	IBM	hsqldb	small
	IBM	luindex	large
	IBM	luindex	small
	IBM	pmd	small
7	Jikes	jython	large
	Jikes	pmd	large
8	Blackdown	jess	s100
	IBM	jython	small
	JRockit	luindex	large
	SUN	jess	s100
9	JRockit	jess	s100
	JRockit	javac	s100
	JRockit	jack	s100
	JRockit	jython	large
	JRockit	pmd	large
10	Jikes	compress	s1
11	Blackdown	hsqldb	large
	IBM	jess	s100
	IBM	hsqldb	large
	IBM	jython	large
	IBM	pmd	large
12	Jikes	jess	s100
	Jikes	mtrt	s100
	Jikes	jython	small
13	Jikes	db	s100
14	Blackdown	bloat	large
	JRockit	antlr	large
	JRockit	bloat	large
	JRockit	hsqldb	large
	JRockit	jython	small
	SUN	bloat	large
15	Blackdown	mpegaudio	s100
	IBM	mpegaudio	s100
	Jikes	mpegaudio	s100
	SUN	mpegaudio	s100
16	JRockit	jess	s1
	JRockit	db	s1
	JRockit	javac	s1
	JRockit	mtrt	s1
	JRockit	jack	s1
	JRockit	antlr	small
	JRockit	bloat	small
	JRockit	fop	large
	JRockit	fop	small
	JRockit	luindex	small
	JRockit	pmd	small
17	Jikes	jess	s1
	Jikes	db	s1
	Jikes	javac	s1
	Jikes	jack	s1
18	IBM	jess	s1
	IBM	db	s1
	IBM	javac	s1
	IBM	mpegaudio	s1
	IBM	mtrt	s1
	IBM	jack	s1
19	SUN	jython	small
20	Blackdown	jess	s1
	Blackdown	db	s1
	Blackdown	javac	s1
	Blackdown	jack	s1
	Blackdown	bloat	small
	Blackdown	fop	small
	Blackdown	pmd	small
21	Blackdown	javac	s100
	Blackdown	mpegaudio	s1
	Blackdown	mtrt	s1
	Blackdown	antlr	small
	Blackdown	hsqldb	small
	IBM	fop	large
22	Blackdown	fop	large
	SUN	fop	large
23	Jikes	mpegaudio	s1
	Jikes	mtrt	s1
	Jikes	antlr	small
24	IBM	compress	s100
	IBM	mtrt	s100
25	Blackdown	luindex	large
	SUN	luindex	large

Table 2.6: Results of a K-means clustering on the all the workload data from the SPECjvm98 and DaCapo suites with the *s1*, *small*, *s100*, and *large* input sets, executed on each of the virtual machines – the first 25 clusters out of 40.

Cluster	VM	Benchmark	Input
26	Blackdown	compress	s100
	Blackdown	compress	s1
	Jikes	compress	s100
	JRockit	compress	s100
	SUN	compress	s100
	SUN	compress	s1
27	IBM	bloat	large
	Jikes	bloat	large
28	SUN	jess	s1
	SUN	db	s1
	SUN	javac	s1
	SUN	mpegaudio	s1
	SUN	mtrt	s1
	SUN	jack	s1
	SUN	antlr	small
	SUN	bloat	small
	SUN	fop	small
	SUN	hsqldb	small
	SUN	pmd	small
29	Blackdown	jython	large
	Blackdown	pmd	large
30	Blackdown	jack	s100
	Blackdown	jython	small

Cluster	VM	Benchmark	Input
31	SUN	javac	s100
	SUN	jack	s100
32	IBM	compress	s1
33	JRockit	db	s100
34	IBM	db	s100
35	JRockit	mtrt	s100
36	SUN	hsqldb	large
	SUN	jython	large
	SUN	pmd	large
37	Blackdown	luindex	small
37	SUN	luindex	small
38	JRockit	compress	s1
	JRockit	mpegaudio	s1
	JRockit	hsqldb	small
39	Jikes	antlr	large
	Jikes	luindex	large
	Jikes	luindex	small
40	Blackdown	antlr	large
	SUN	antlr	large

Table 2.7: Result of a K-means clustering on the all the benchmark data from the SPECjvm98 and DaCapo suites with the *s1*, *small*, *s100*, and *large* input sets, executed on each of the virtual machines – the last 15 clusters out of 40.

picking a certain benchmark subset. Yet the above technique provides a way to choose a subset with some degree of confidence that the workload space will be well covered, and that the probability of obtaining wrong results is limited. However, we can but agree with the findings of Pérez et al. [86] that subsetting is inherently dangerous and thus we do recommend using the full set of available benchmarks when doing final performance measurements.

2.5.5 Comments on the garbage collector

As noted in Section 2.3, the choice of the garbage collector was not consistent, i.e., different virtual machine configurations have different garbage collectors. In an experiment to measure the effect of the garbage collector, we use five collection algorithms in the Jikes RVM: CopyMS, GenCopy, GenMs, MarkSweep, and SemiSpace. In Figures 2.18 and 2.19, we show the resulting dendrogram for the benchmarks run with their largest input set. This figure shows that the workloads are quite well mixed by garbage collector, and as such, that the choice of GC algorithm in the study presented in this chapter is of lesser importance, i.e., it does not change any of the conclusions.

2.6 Related work

Before we embarked on the study described in this chapter, other researchers had already contributed considerably to the understanding of the behaviour of Java applications. Here we present a brief summary of some of this work.

Bowers and Kaeli [26] characterise the SPECjvm98 benchmarks at the bytecode level. They conclude that Java applications have a large number of loads in their dynamic bytecode stream.

Hsieh et al. [57] compare the performance of the SUN JDK 1.0.2 Java interpreter, a bytecode to native code translator called Caffeine [58] and a compiled C/C++ version of the code. This is done based on simulations. They conclude that the interpreter exhibits poor branch target buffer (BTB) performance, poor I-cache behaviour and poor D-cache behaviour compared to the other approaches.

Chow et al. [33] compare Java workloads with non-Java workloads (e.g., SPEC CPU95, SPEC CINT95, etc.) using principal components analysis. In this study, the authors focus on the branch behaviour, i.e., the number of conditional jumps, direct calls, indirect calls, indirect jumps, returns, etc. Based on simulation results, they conclude that Java workloads appear to have more indirect branches than non-Java workloads. However, the number of indirect branch targets can be small. For example, when considering the number of indirect target changes, Java workloads are no worse than some SPEC CINT95 benchmarks. Our study differs from this work in the following aspects: (i) we use more virtual machines, (ii) they consider only branch characteristics, and (iii) their goal was to compare Java workloads versus non-Java workloads, whereas we seek to gain insight in the interaction between the components comprising a Java workload.

Radhakrishnan et al. [87, 88] analyse the behaviour of the SPECjvm98 benchmarks by instrumenting the virtual machines and by simulating execution traces. They used two virtual machines: the Sun JDK 1.1.6 and Kaffe 0.9.2. They conclude that (i) 45 out of the 255 bytecodes constitute 90% of the dynamic bytecode stream, (ii) an oracle translation scheme (optimal translation selection) in case of a JIT compiler can improve performance by only 10% to 15%, (iii) the I-cache and D-cache performance is better for Java applications than for C/C++ applications, except for the D-cache in JIT mode, (iv) write misses due to installing JIT compiler output have a significant impact on the D-cache performance in JIT mode, and (v) the amount of ILP is higher under JIT mode than under interpreter mode.

Li et al. [69] characterise the behaviour of SPECjvm98 Java benchmarks through complete system simulation. This was done by using the Sun JDK 1.1.2 virtual machine and the SimOS complete system simulator [90]. They conclude that the SPECjvm98 applications (on s100) spend on average 10% of their time in system (kernel) activity compared to only 2% for the four SPEC CINT95 benchmarks studied. Generally, the amount of time in kernel activity

is higher for the JIT compiler mode than for the interpreter mode. The kernel activity is mainly due to TLB miss handler invocations. Also, they conclude that the SPECjvm98 benchmarks have inherently poor instruction-level parallelism (ILP) compared to other classes of benchmarks.

In [70], Li et al. analyse the impact of kernel activity on the branch behaviour of Java workloads. They conclude that branches in OS code exhibit a different biased behaviour which increases the branch misprediction rate significantly. As such, they propose OS-aware branch prediction schemes which outperform conventional branch predictors.

Shuf et al. [99] characterise the memory behaviour of Java workloads. They conclude that some SPECjvm98 benchmarks are not truly object-oriented and are thus not representative for real Java workloads. As such, they propose to use the server-oriented pBOB benchmark [11] in studies on Java workloads in addition to some SPECjvm98 benchmarks. Secondly, they conclude that the number of hot spots is small for most Java programs. Consequently, expensive algorithms are justified for run-time optimisations. Third, they conclude that the D-cache behaviour of Java workloads is poor resulting in high D-cache miss rates – even fairly large L2 caches do not increase performance significantly. In addition, they conclude that the TLB as well as the cache behaviour is worse for Java workloads than for technical benchmarks, but comparable to commercial workloads.

After the work presented here was published at OOPSLA 2003, other researchers have extended the insights we obtained. Blackburn et al. [17] confirmed our findings and show that a Java performance evaluation methodology, next to considering multiple JVMs, should also consider multiple heap sizes as well as multiple hardware platforms. Choosing a particular heap size and/or a particular hardware platform may draw a fairly different picture and may even allow to game results and conclusions made thereof.

2.7 Conclusions

In this chapter, we studied the relationship between the behaviour of a Java workload and the virtual machine, the benchmark and the input set which comprise the workload. From the experiments we conducted, we can draw the following conclusions:

- For the *s1* input set of SPECjvm98 and the *small* input set of DaCapo, the behaviour as observed at the microarchitectural level is mainly determined by the virtual machine. This is due to the fact that small input sets lead to short-running benchmarks. This causes the start-up of the virtual machine and the initial compilation of frequently executed methods to be the largest contributor to the overall behaviour. As such, this suggests that using the *s1* or *small* input sets in a Java system performance analysis might not be good practice (unless one is mainly interested in

measuring start-up time) since the results that are obtained from such an analysis can be highly biased by the virtual machine that is used.

- Using the short-running input set as a representative for the long-running input set for both SPECjvm98 and DaCapo is clearly not good practice, since the behaviour that is observed at the microarchitectural level can be quite different for both input sets. One reason obviously is the fact that a virtual machine has more opportunities for run-time optimisations for long-running benchmarks than for short-running benchmarks.

- With large(r) input sets, the applications have the most impact on overall behaviour, but the virtual machine still exerts a significant influence.

- In general, researchers should be careful when reporting results using only one or two virtual machines. The results presented in this chapter clearly show that the behaviour that is observed at the microarchitectural level is highly dependent on the virtual machine. As such, results obtained for one virtual machine might not be transferable for another virtual machine and vice versa.

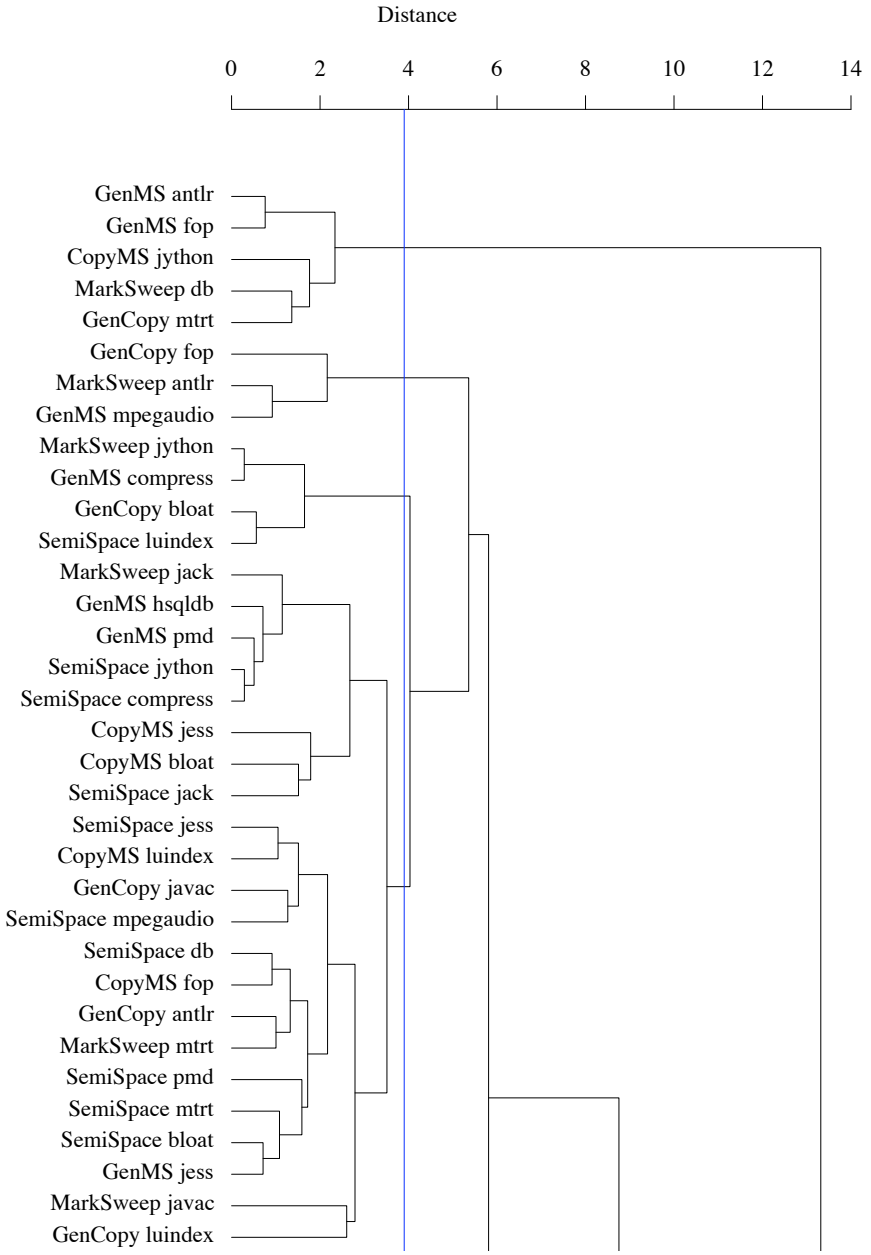


Figure 2.18: Top part of the dendrogram showing the impact of the garbage collector on Java workload behaviour using five collectors in the Jikes RVM. All benchmarks were run with their largest input set.

Figure 2.19: Bottom part of the dendrogram showing the impact of the garbage collector on Java workload behaviour using five collectors in the Jikes RVM. All benchmarks were run with their largest input set.

Chapter 3

Prevalent performance analysis approaches

It is not worth an intelligent man's time to be in the majority. By definition, there are already enough people to do that. – **G. H. Hardy**

In the first chapter, we touched upon the fact that Java workload performance is non-deterministic. This results in an execution time that exhibits quite some variation across multiple executions of the same workload. Obviously, this needs to be taken into account when reporting the performance of Java applications. In the past, researchers have come up with several approaches to set up their experiments and to report performance numbers of Java applications, such as reporting best performance, second-best performance, average performance, or even worst performance from a number of benchmark executions. In this chapter, we give an overview of these commonly used approaches. We make a distinction between experimental design and data analysis. In the next chapter, we then show which consequences these approaches may have for the conclusions reached based on poor data analysis.

3.1 Key aspects of benchmarking

Benchmarking is at the heart of experimental computer science research and development. Market analysts compare commercial products based on published performance numbers. Developers benchmark products under development to assess their performance. And researchers use benchmarking to evaluate the impact on performance of their novel research ideas. As such, it is absolutely crucial to have a rigorous benchmarking methodology, that is

generally accepted. Essentially, there are two key aspects that contribute to a good benchmarking strategy: (i) experimental design, and (ii) data analysis. Additionally, we believe any usable benchmarking strategy should reflect reality, i.e., people should be able to use the benchmarking results as a foundation on which they make design and purchasing decisions.

Example 3.1. *Consider the following situation. A service provider has to decide which machine to buy in order to serve his customers' – and by extension, his own – interests best. Rather than tackling the task of setting up a number of experiments for which he must acquire and configure a test machine from each vendor, and moreover spend quite some time measuring performance, he looks for available data that can help him make a well-informed decision. Googling leads him to the SPEC website where he has access to a large number of performance results for a number of platforms. The question remains if this data will steer him towards the correct purchasing decision, i.e., do the published results reflect his (real-life) situation or are they artificial due to either a poor experimental setup and/or poor data analysis?*

Experimental design refers to setting up the experiments to be run and requires a good understanding of the system being measured. It should be determined prior to conducting the experiment, with the limitations and peculiarities of the system in mind. In particular, it is challenging to decide on a good experimental design for managed runtime systems, such as the Java platform, because there are several factors affecting overall performance. Said factors are of lesser concern when benchmarking classical compiled programs, as can be found in the SPEC CPU suites, for example, programs written in C, C++ and Fortran. The execution of these programs does not use JIT-compilation or sampling to steer runtime optimisations. Consequently, the performance suffers much smaller variability. At this moment, there is a growing awareness of this difficulty in quantifying managed runtime system performance. The past few years, researchers have been publishing a number of papers that reveal the complex interactions between low-level events and overall performance for Java workloads [15, 51, 54, 75, 108]. As we showed in the previous chapter, the importance of a well chosen and motivated experimental design should not be underestimated. Once again we stress that one should carefully consider the benchmarks, the virtual machine(s), the inputs, and the hardware platform chosen in the setup [17]. Indeed, not appropriately considering and motivating one of these key aspects, or not appropriately describing the context within which the results were obtained and how they should be interpreted may give a skewed view. Good scientific practice dictates experiments should be repeatable[1], and detailing the experimental design helps to achieve that goal.

Orthogonal to the experimental design aspect, there is the matter of data analysis. This aspect specifies how to analyse and report results obtained from the experiments, preferably in a trustworthy manner. Specifically, in perfor-

[1] In computer science this is notoriously hard to achieve, precisely because the experimental setup is rarely provided in sufficient detail.

mance evaluation studies, the data analysis should be able to deal with any phenomenon that occurs as a consequence of the experimental design. With Java – and managed runtime systems in general – one of the major phenomena to deal with is non-determinism in the measurements – for almost any experimental design. Of course, this is also true for applications written in for example C, but in a managed runtime environment, there is a higher probability some non-determinism will perturb the measurements.

3.2 Causes of non-determinism

As mentioned, there are several sources of non-determinism in the virtual machine that can affect runtime behaviour and thus execution time. The Just-In-Time (JIT) compilation system, and the associated optimisation system are a potential sources of non-determinism. Frequently, a virtual machine will employ some sampling mechanism to drive optimisations by which good performance can be attained. Most of the production virtual machines in use today use a timer-based sampling approach [30]. During the execution of Java applications, hot methods are determined by sampling the method(s) at the top of the execution stack at regular times. This is done by a (maskable) timer interrupt, which is passed to the virtual machine by the OS as a signal. However, there is a variable delay from when the OS discovers that a thread (in this case, a VM process) needs to be notified using this signal and when it schedules the thread on the CPU, so it can initiate the sample taking. The samples directly influence the decisions made by the optimisation framework of a virtual machine. If a method is sampled a sufficient number of times, it will be proposed for optimisation. Because optimised methods execute faster, the point in time at which a method gets optimised is reflected in the total execution time for the workload. However, across different executions, the same method M may be proposed either at another time, or even not at all. Thus, a run where M is optimised early will finish sooner compared to a run where M is optimised later. Yet, both runs will perform the same task[2].

Example 3.2. *Consider the following experiment. We execute 30 invocations of the Jikes RVM for each benchmark using the GenMS garbage collector with a variable heap size of maximally 256 MB. In each of these runs, we keep track of the samples taken per method. Using the weighted overlap metric described in [30], we compare each of the 30 runs to determine the stability of the sampling. Briefly put, we take the element-wise minimum of two weighted vectors representing the number of samples taken per method. The weighing is done by normalising the vector from each run by its sum. For example, consider the sample count vectors* $a = (3, 0, 1)$ *and* $b = (2, 1, 2)$*. In this case the weighted vectors become* $\frac{a}{|a|} = (3/4, 0, 1/4)$ *and* $\frac{b}{|b|} = (2/5, 1/5, 2/5)$*. The resulting vector is then* $(2/5, 0, 1/4)$*, the column-sum of which equals 65%.*

[2] That is, provided no time-dependent code-paths are present in the application itself.

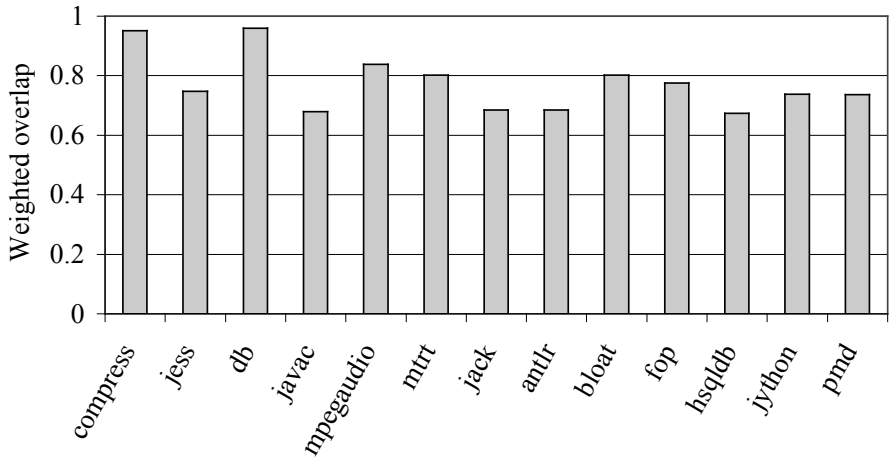


Figure 3.1: The average weighted overlap for 30 runs using a single VM invocation and a single benchmark iteration in each run. The experiments were conducted on the Jikes RVM using the GenMS garbage collector and a variable heap size with a maximum of 256MB.

Figure 3.1 shows the average overlap per benchmark. For some benchmarks this overlap is as low as 67.4%. The global average is 75.8%. This means that for almost 25% of the samples taken, the sampled method differs widely per run.

Another source of non-determinism is thread scheduling in time-shared and multiprocessor systems. Running multithreaded workloads, as is the case for most Java programs, requires thread scheduling in the operating system and/or virtual machine[3]. Different executions of the same program may introduce different thread schedules, and may result in different interactions between threads, affecting overall performance. The non-determinism introduced by JIT compilation and thread scheduling may affect the points in time where garbage collections occur. Garbage collection in its turn can affect program locality, and thus memory system performance as well as overall system performance. Yet another source of non-determinism are various system effects, such as system interrupts – this is not specific to managed runtime systems: it is a general concern when running experiments on real hardware.

3.3 Runtime variability

As Figure 1.4 demonstrates, start-up run-time can exhibit quite some variability. In Figure 3.2, for the *db* benchmark, we observe a tendency of decreasing

[3]The latter is often referred to a green threading, in which multiple Java threads are scheduled by the VM onto a single native or POSIX thread, and no OS intervention is required.

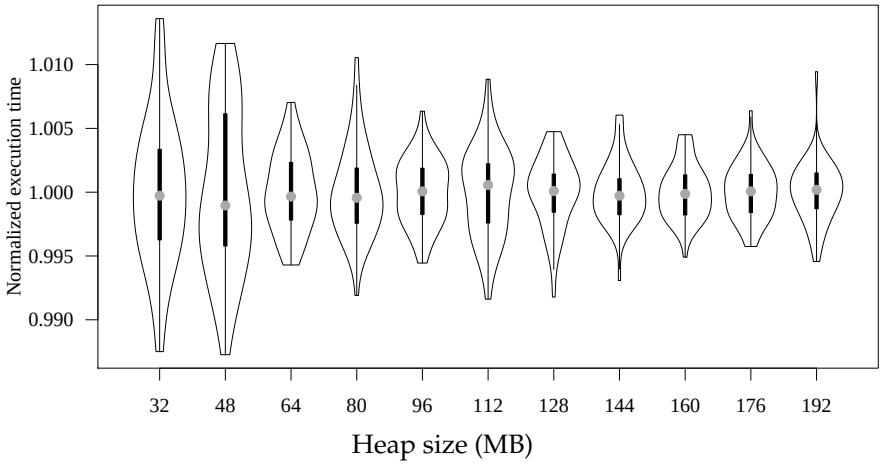


Figure 3.2: Run-time variability normalised to the mean execution time for start-up performance of *db*. These results assume 30 VM invocations on the AMD Athlon platform with the GenMS collector for various heap sizes, ranging from the minimal heap size up to six times that amount.

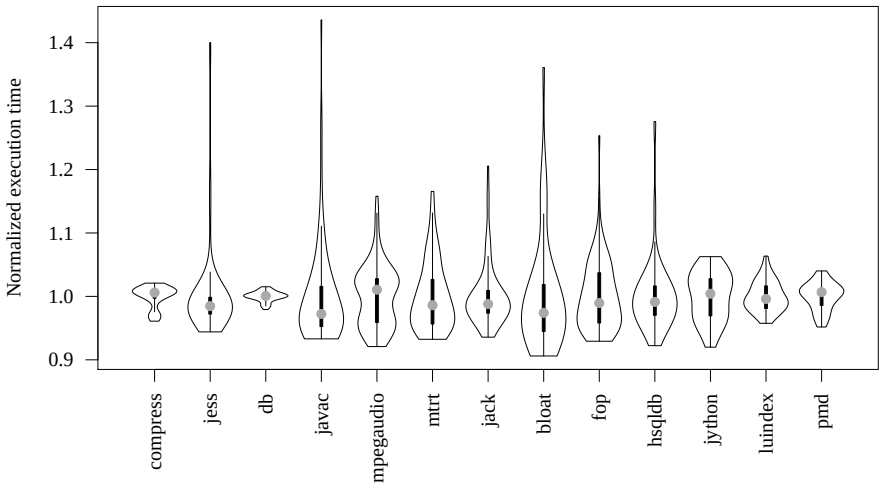


Figure 3.3: Run-time variability normalised to the mean execution time for steady-state performance. These experiments assume 10 VM invocations and 30 benchmark iterations per VM invocation on the AMD Athlon platform with the GenMS collector and a per-benchmark heap size that is twice as large as the minimal heap size reported in Table 2.2. The dot represents the median.

variability with increasing heap size. We observe similar results for the other benchmarks. This suggests that the garbage collection has a large impact on overall variability: a smaller heap size implies more collections, and this coincides with larger variability.

Figure 3.3 shows a similar trend for steady-state behaviour. All measurements were done on the AMD Athlon platform using the Jikes RVM[4]. For the start-up graph, the experiment uses 30 VM invocations. For the steady-state graph, the experiment uses 10 VM invocations and 30 benchmark iterations per VM invocation, of which the last 10 were retained. The execution times in the graphs were normalised to have a unit mean. Once again, we observe fairly significant runtime variability, even during the steady-state execution. We mentioned in the introductory chapter that the CoV is generally around 2%. The performance difference between the maximum and the minimum performance number for steady-state varies across the benchmarks, but is around 20%.

3.4 Prevalent methodologies

To learn which approaches are commonly used today, we conducted a survey, examining 50 papers. The papers published in the last seven years (from 2000 onward) at top-tier conferences – such as Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), Programming Language Design and Implementation (PLDI), Virtual Execution Environments (VEE), Memory Management (ISMM) and Code Generation and Optimization (CGO) – exhibit a wide range of approaches in both experimental design and data analysis.

Surprisingly enough, about one third of the papers (16 out of the 50 papers) do not specify the methodology used in the paper. This not only makes it difficult for other researchers to reproduce the results presented in the paper, it also makes understanding and interpreting the results hard, if not outright impossible. This was especially the case for papers published prior to 2003. In sync with the growing awareness [17, 43] of having a rigorous performance evaluation methodology, later papers often have a more detailed description of their methodology.

In spite of these recent advances towards a rigorous Java performance benchmarking methodology, there is no consensus among researchers on what methodology to use. In fact, almost all research groups come with their own methodology. We now discuss some general features of these prevalent methodologies and subsequently illustrate these using a number of example methodologies.

[4]We used the SVN head version of February 12, 2007

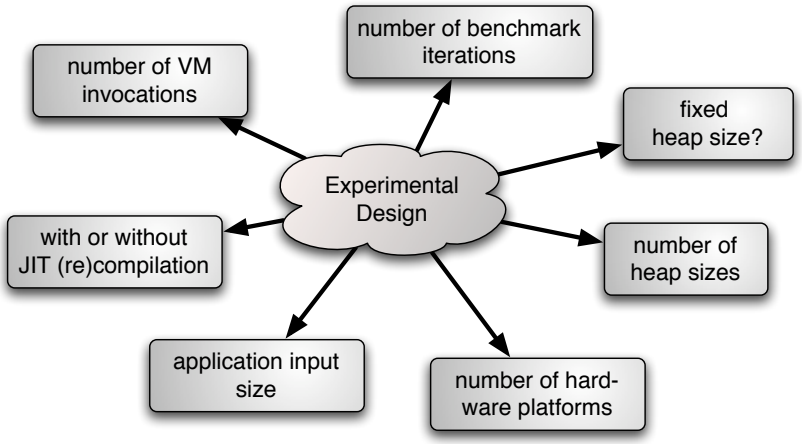


Figure 3.4: Choices in experimental design for Java performance analysis.

3.4.1 Experimental design

Figure 3.4 shows an overview of the choices a researcher must make when setting up an environment to measure the performance of a Java application. This involves making choices regarding the physical and virtual machine(s), the benchmarks, the execution mode (start-up or steady-state), the heap size (fixed vs. variable), etc. The following paragraphs discuss these choices in more detail.

Start-up execution versus steady-state execution

Although the terms used here are loosely defined, there is a substantial difference between start-up execution and steady-state execution. In the former case, the application execution is intermingled with the virtual machine execution on behalf of the application to load classes, to compile and to optimise methods, etc. In steady-state, no more (or at least, little) optimisation is done, and the virtual machine only interferes with the execution when garbage needs to be collected or a thread switch occurs. To model steady-state behaviour with benchmarks that do not nearly run long enough to actually reach such a state, it is possible to iterate the same benchmark multiple times. As we mentioned earlier, both SPECjvm98 and DaCapo offer a harness that allows this. Therefore, for start-up execution benchmarkers usually consider a single iteration, whereas for steady-state they consider the performance after a number of iterations have passed.

One VM invocation versus multiple VM invocations

To gather a number of measurements in the start-up performance scenario, a benchmarker executes multiple virtual machine invocations, because he can only derive a single performance number per invocation. In the steady-state scenario however, a benchmarker can get performance numbers from subsequent iterations once steady-state has been reached. Therefore, he might restrict himself to a single virtual machine invocation to save time[5].

Including compilation versus excluding compilation

Some researchers report performance numbers that include JIT compilation overhead, while others report performance numbers excluding JIT compilation overhead. In a managed runtime system, JIT (re)compilation is performed at run-time, and by consequence, becomes part of the overall execution. Some researchers want to exclude JIT compilation overhead from their performance numbers in order to isolate Java application performance and to make the measurements (more) deterministic, i.e., have less variability in the performance numbers across multiple executions.

A number of approaches have been proposed to exclude compilation overhead. One approach is to compile all methods executed during a first execution of the Java application, i.e., all methods executed are compiled to a predetermined optimisation level, in some cases the highest optimisation level. The second run, which is the timing run, does not do any compilation. Another approach, which is becoming increasingly popular, is called *replay compilation* [59, 92], which is used in 7 out of the 50 papers in our survey. Under replay compilation, a first run is used to determine which methods have been optimised. During the second run, these particular methods are compiled in a first benchmark iteration. In a second iteration, (adaptive) compilation is switched off, and performance is measured.

Forced GCs before measurement

Some researchers perform a full-heap garbage collection before doing a performance measurement. This reduces the non-determinism observed across multiple iterations due to garbage collections kicking in at different times across different VM invocations.

Other considerations

Other considerations concerning the experimental design include one hardware platform versus multiple hardware platforms; one heap size versus mul-

[5]Note that we are not saying this is good practice!

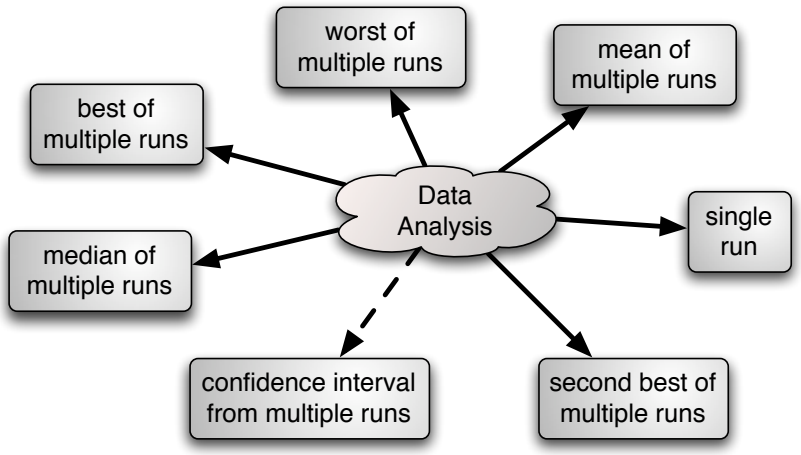


Figure 3.5: Prevalent methodologies used for analysis of experimental Java performance data.

tiple heap sizes; a single VM implementation versus multiple VM implementations.

An important consideration is the choice between back-to-back measurements – 'aaabbbccc' (a, b and c represent benchmarks) – versus interleaved measurements – abcabcabc'. In the former case, we execute the same workload a number of times, say n , after which we change the configuration, e.g., use a different benchmark, and redo the experiment again n times, and so on. In the latter case, we switch configurations after each run.

3.4.2 Data analysis

When the experiments have been run, we need to process the measurements such that conclusions can be drawn: the data analysis step. Figure 3.5 gives an overview of prevalent data analysis techniques we encountered in our survey.

Average or median versus best versus worst run

Some methodologies report the average or median execution time across a number of runs – typically more than 3 runs are considered; some go up to 50 runs. Others report the best or second best performance number, and yet others report the worst performance number.

The SPEC run rules for example state that SPECjvm98 benchmarkers must run their Java application at least twice, and report both the best and worst of all runs. The rules seem to imply though that the benchmarks are run for

several iterations in a single VM invocation. However, both best and worst approaches are used by several papers for start-up execution [4, 5, 17, 20, 59, 103, 111]. The intuition behind the worst performance number in the SPEC rules is to report a performance number that represents program execution intermingled with class loading and JIT compilation. In a more general context, a worst case execution can be useful to determine if for example real-time deadlines can be met. The intuition behind the best performance number – as outlined in the SPEC rules – is to report a performance number where overall performance is mostly dominated by program execution, i.e., class loading and JIT compilation are less of a contributor to overall performance and steady-state regime has taken over. In general, the motivation for reporting a *best* performance number is that if a new technique beats the best run, it is obviously worthwhile.

The most popular approaches are average and best – 8 and 10 papers out of the 50 papers in our survey, respectively; median, second best and worst are less frequent, namely 4, 4 and 3 papers, respectively.

Confidence intervals versus a single performance number

In only a small minority of the research papers (4 out of 50), confidence intervals are reported to characterise the variability across multiple runs. The others papers report but a single performance number.

3.5 Replay compilation

Replay compilation [59, 92] is a recently introduced experimental design methodology that fixes the compilation/optimisation load in a Java virtual machine execution. As mentioned before, the motivation is to control non-determinism, and, by doing so, facilitate performance analysis.

3.5.1 Basic replay compilation mechanism

Replay compilation requires a *profiler* and a *replayer*. The profiler, see Figure 3.6, records the profiling information used to drive the compilation decisions, e.g., edge counts, path and dynamic call graph info, etc., as well as the compilation decisions, e.g., method M_1 was compiled at optimisation level 0, method M_2 was compiled at optimisation level 2, etc. Typically, researchers run multiple experiments yielding multiple profiles, and a single *compilation plan* is then determined from these profiles.

In the replay phase, the compilation plan is read when the virtual machine starts up. Each method is compiled to the optimisation level specified in the compilation plan upon its first invocation. Consequently, after the first iteration, all methods have been optimised according to the plan. This also means

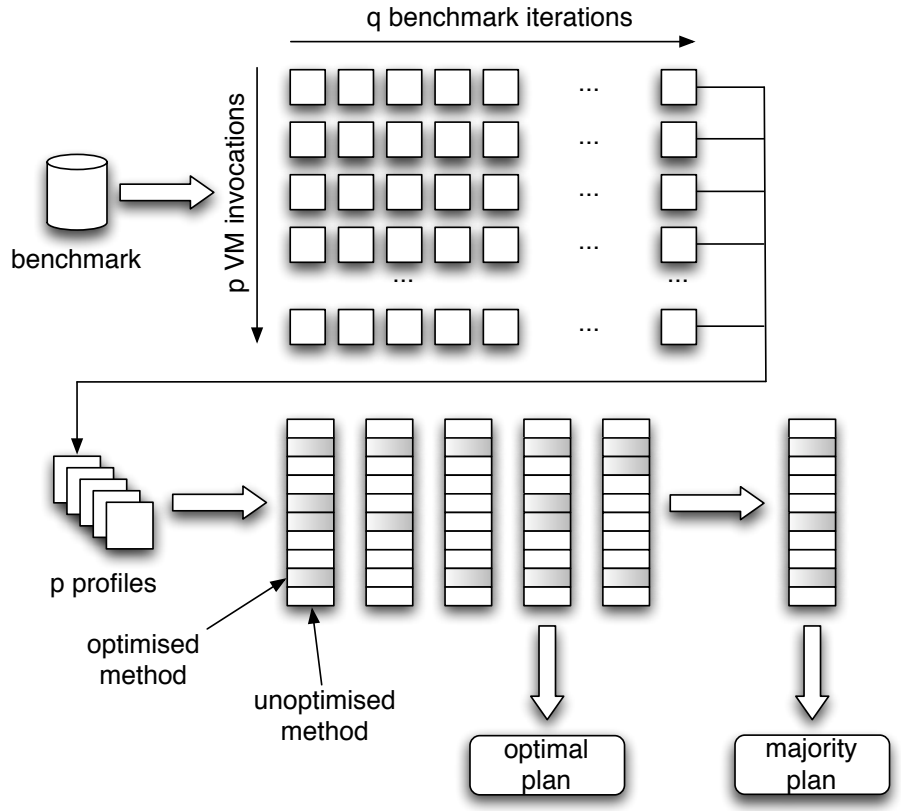


Figure 3.6: The profiling phase for a typical replay compilation setup. Up to now, all examined papers either use $q = 1$ or do not specify any particular value for q. Current work uses a value for p that is usually smaller than 10. The optimal profile corresponds to the fastest run; the majority profile is a combination of all assembled profiles.

that the compilation/optimisation load is fixed during this iteration, which facilitates performance analysis. The replay phase typically consists of two benchmark iterations. The first iteration, described above, includes compilation overhead according to the plan, and is referred to as the *mix* run. In the second iteration the adaptive (re)compilation is turned off – this is called the *stable* run. Usually, the stable run is used as the timing run. In order to remove dependencies between the mix and stable run caused by dead objects and the associated perturbation during GC, a full collection is typically done between both iterations.

3.5.2 Design options for replay

Researchers typically select a single compilation plan from a number of profiles, or, alternatively, combine these profiles into a single compilation plan. Some researchers pick the best profile as the compilation plan, i.e., the profile that yields the best overall performance, see for example [15, 18, 19, 23]. Others select the median optimisation level observed across a number of profiles for each method [110]. Yet others select the methods that are optimised in the majority of the runs, and set the optimisation level for the selected methods at the highest optimisation levels observed in the majority of the runs, see for example [32, 59, 93, 92]. Finally, some researchers select the methods observed in the intersection of multiple profiles [101].

Another design option involves choosing the moment at which to collect the profiles. A benchmark can be iterated multiple times within a single VM invocation. When this happens, more and more methods will be selected for JIT optimisation, i.e., the code quality will steadily improve as more and more methods get optimised to higher levels of optimisation. The question then is when to collect the profile across these multiple benchmark iterations. One option could be to finish the profile collection after the first benchmark iteration as is commonly done in current practice. Another option would be to collect the profile across multiple benchmark iterations. This will result in a profile that represents better code quality.

The final design option is how to configure the system setup (virtual machine configuration, garbage collection strategy, heap size, etc.) when collecting the profiles.

3.5.3 Issues

A single compilation plan

Current practice in replay compilation considers a single compilation plan during replay. As we will show in Chapter 5, this can be misleading. The reason is that a single compilation plan does not account for the variability observed in compilation load across multiple runs under non-deterministic VM executions. We therefore advocate using multiple compilation plans at replay time. This is consistent with the work we present in Chapter 4 on using statistical data analysis for coping with non-determinism [50], which advocates using an average performance number along with a confidence interval computed from a number of benchmarking experiments instead of picking a performance number from a single experiment.

Non-determinism

Replay compilation, although it controls non-determinism to a large extent, does not completely eliminate non-determinism. There are a number of re-

maining sources of non-determinism that replay compilation does not control. For example, replay compilation does not control thread scheduling. Different thread scheduling decisions in time-shared and multi-threading environments across different runs of the same experiment can affect performance. For example, different thread schedules may lead to different points in time where garbage is being collected leading to different data layouts, which may affect memory system performance as well as overall performance. Also, various system effects, such as interrupts, introduce non-determinism when run on real hardware. In Chapter 5, we present recommendation on how to deal with non-determinism in the case of replay compilation. Essentially, by controlling non-determinism, replay compilation reduces the required number of measurements to reach a certain level of confidence in the results.

Replay compilation as experimental design

Replay compilation is an experimental design choice that may be appropriate for some experiments but inappropriate for others. It is up to the experimenter, who has a good understanding of the system under measurement, to determine whether replay compilation is appropriate or not. Specifically, the implicit assumption for replay compilation is that the innovation under evaluation does not affect compilation decisions, i.e., the compiler/optimiser is assumed to make the same compilation/optimisation decisions irrespective of the innovation under evaluation. This may or may not be a valid assumption depending on the experiment at hand.

3.5.4 Use-case scenarios

There are several use-case scenarios for which replay compilation is a useful experimental design setup. We enumerate several cases as they are in use today – this enumeration illustrates the wide use of replay compilation as an experimental design setup for managed runtime systems.

JIT innovation

JIT research, such as compiler/optimiser innovation, may benefit from replay compilation as an experimental setup. Researchers evaluating the efficacy of JIT innovation want to answer questions such as 'How does my innovation improve application code quality?', and 'What's the compile time overhead that the innovation adds?' The problem at hand is that in a virtual execution environment with dynamic compilation, application code execution and compilation overhead are intermingled. The question then is how to tease apart the effect that the JIT innovation has on code quality and compile time?

Replay compilation is a methodology that enables teasing apart code quality and compile time overhead, see for example Cavazos and Moss [32], who

study compiler scheduling heuristics. The mix run provides a way of quantifying the overhead the innovation has on compilation time. The stable run provides a way of quantifying the effect of the innovation on code quality.

Innovation in profiling or JVM innovations

A research topic that is related to JIT innovation is profiling, i.e., an improved profiling mechanism provides a more accurate picture for analysis and optimisation. In [24], Bond and McKinley use replay compilation to gauge the performance of their probabilistic calling context implementation. Because they fix the compilation in the first iteration, the execution of the second iteration can be used for performance and overhead measurement. Similarly, in [22], Bond and McKinley determine the amount of overhead their technique incurs, by comparing non-deterministic runs with both mix and stable replay execution. Schneider et al. [96] use execution replay in their experiments, where they show how hardware performance monitors can drive online optimisations, for example in a generational garbage collector that reduces L1 cache misses by 21%.

GC innovation

Garbage collection (GC) research can also benefit from replay compilation. In fact, many recent garbage collection research papers use replay compilation as their experimental design methodology [15, 18, 19, 53, 59, 92, 101, 93, 110]. The reason why replay compilation is useful for garbage collection research is that it fixes the compilation load, and by doing so, it controls non-determinism which facilitates the comparison of garbage collection design alternatives.

Other applications

There exist a number of other applications to replay compilation. Krintz and Calder [65] for example annotate methods with analysis information collected offline, similar to a compilation plan. These annotations significantly reduce the time to perform dynamic optimisations. Ogata et al. [83] use replay compilation to facilitate the debugging of JIT compilers.

3.6 Example methodologies

To demonstrate the diversity in prevalent Java performance evaluation methodologies, both in terms of experimental design and data analysis, we refer to Table 3.1 which summarises the main features of a number of example methodologies. This wide diversity in methodologies around today illustrates the growing need for a rigorous performance evaluation methodology;

These example methodologies are among the most rigorous methodologies observed during our survey: these researchers clearly describe and/or motivate their methodology whereas many others do not.

We illustrate the aforementioned choices in more detail in the following three examples.

Example 3.3. *McGachey and Hosking [78] (methodology B in Table 3.1) iterate each benchmark 11 times within a single VM invocation. The first iteration compiles all methods at the highest optimisation level. The subsequent 10 iterations do not include any compilation activity and are considered the timing iterations. Only the timing iterations are reported; the first compilation iteration is discarded. And a full-heap garbage collection is performed before each timing iteration. The performance number reported in the paper is the average performance over these 10 timing iterations along with a 90% confidence interval.*

Example 3.4. *Arnold et al. [4, 5] (methodologies F and G in Table 3.1) make a clear distinction between start-up and steady-state performance. They evaluate the start-up regime by timing the first run of a benchmark execution with a medium input set (s10 for SPECjvm98). They report the minimum execution time across five benchmark executions, each benchmark execution involving a new VM invocation. For measuring steady-state performance, in [4], Arnold et al. report the minimum execution time across five benchmark executions with a large input set (s100 for SPECjvm98) within a single VM invocation. Arnold et al. [5] use a different methodology for measuring steady-state performance. They do 10 experiments where each benchmark runs for approximately 4 minutes; this results in 10 times N runs. They then take the median execution time across these 10 experiments, resulting in N median execution times and then report the minimum median execution time. All the performance numbers reported include JIT compilation and optimisation, as well as garbage collection activity.*

Example 3.5. *The third example methodology uses replay compilation to drive the performance evaluation [15, 17, 59, 92]. They select the best compilation plan, i.e., the plan that results in the fastest profiling run. The benchmarking experiment then proceeds as follows: (i) the first benchmark run performs compilation using the compilation plan, (ii) a full-heap garbage collection is performed, and (iii) the benchmark is run a second time with adaptive optimisation turned off. This entire process is done m times, and the best run is reported. Reporting the first benchmark run is called the mix method; reporting the second run is called the stable method.*

3.7 Conclusions

It is abundantly clear that there is a broad range of methodologies in use today for measuring Java performance. Both experimental design and data analysis seem to be tailored to the need of the researcher. Having such diversity makes it difficult to objectively compare results across research studies. Furthermore,

as we will show in the next chapter, none of the most widely used techniques adequately deal with non-determinism, though they mostly recognise its presence.

	Methodology reference	A [14]	B [78]	C [94]	D [100]	E [111]	F [4]	G [5]	H [92]	I [17, 59]	J [103]	K [20]	L [10]	M [21]
data analysis	average performance number from multiple runs	✓	✓	✓									✓	
	median performance number from multiple runs							✓						
	best performance number from multiple runs					✓	✓	✓		✓	✓	✓		
	second best performance number from multiple runs													✓
	worst performance number from multiple runs										✓			
	confidence interval from multiple runs		✓		✓									
experimental design	one VM invocation, one benchmark iteration												✓	
	one VM invocation, multiple benchmark iterations		✓	✓			✓					✓		
	multiple VM invocations, one benchmark iteration					✓	✓							✓
	multiple VM invocations, multiple benchmark iterations				✓			✓						
	including JIT compilation			✓			✓	✓		✓				
	excluding JIT compilation		✓			✓			✓	✓				
	all methods are compiled before measurement		✓			✓						✓		
	replay compilation								✓	✓				
	full-heap garbage collection before measurement		✓							✓				
	a single hardware platform	✓	✓	✓		✓	✓	✓	✓		✓	✓		✓
	multiple hardware platforms				✓					✓			✓	
	a single heap size						✓	✓			✓			
	multiple heap sizes	✓	✓	✓	✓	✓			✓	✓		✓	✓	✓
	a single VM implementation	✓	✓	✓	✓	✓	✓	✓	✓			✓	✓	✓
	multiple VM implementations									✓	✓			
	back-to-back measurements													
	interleaved measurements									✓				

Table 3.1: Characterising prevalent Java performance evaluation methodologies (in the columns) in terms of a number of features (in the rows): the '✓' symbol means that the given methodology uses the given feature; the absence of the '✓' symbol means the methodology does not use the given feature, or, at least, the feature is not documented.

Chapter 4

Java performance analysis in the presence of non-determinism

God does not play dice with the universe. – **Albert Einstein**

In the previous chapter, we illustrated that there is a plethora of prevalent approaches, both in experimental design and data analysis for benchmarking Java performance. We now show that there is a pitfall associated with the aforementioned prevalent approaches. In particular, we show that these approaches can lead to misleading or even incorrect conclusions. The reason for this is that the data analysis used in prevalent approaches is not statistically rigorous. In this chapter we present and advocate using a statistically rigorous data analysis approach for both start-up and steady-state performance.

4.1 Introduction

The pitfall in using the *best-of* prevalent method was clearly illustrated in Figure 1.5 on page 13, which compares the execution time for running Jikes RVM with five garbage collectors (CopyMS, GenCopy, GenMS, MarkSweep and SemiSpace) for the SPECjvm98 *db* benchmark with a 120 MB heap size, see also Example 1.3 on page 12. To recapitulate, in 3 out of the 10 comparison, the *best-of* approach leads to a conclusion different from what a rigorous analysis yields.

In this chapter, we present the following contributions:

- We demonstrate that there is a major pitfall associated with today's prevalent Java performance evaluation methodologies, especially in terms of data analysis. The pitfall is that they may yield misleading and even incorrect conclusions. The reason is that the data analysis employed by these methodologies is not statistically rigorous.

- We advocate adding statistical rigour to performance evaluation studies of managed runtime systems, and in particular Java systems. The motivation for statistically rigorous data analysis is that statistics, and in particular confidence intervals, enable one to determine whether differences observed in measurements are due to random fluctuations in the measurements or due to actual differences in the alternatives compared against each other. We discuss how to compute confidence intervals and discuss techniques to compare multiple alternatives.

- We advocate the following approaches. For start-up performance, we advise: (i) to take multiple measurements where each measurement comprises one VM invocation and a single benchmark iteration, and (ii) to compute confidence intervals across these measurements. For steady-state performance, we advise: (i) to take multiple measurements where each measurement comprises one VM invocation and multiple benchmark iterations, (ii) in each of these measurements, to collect performance numbers for different iterations once performance reaches steady-state, i.e., after the start-up phase, and (iii) to compute confidence intervals across these measurements (multiple benchmark iterations across multiple VM invocations).

This chapter is organised in four main parts. First, we present general statistics theory in Section 4.2, and how it applies to Java performance analysis. Second, in Section 4.3, we outline a rigorous approach for analysing both start-up and steady-state execution. Third, in Sections 4.4 and 4.5, we exhibit the pitfall under discussion more in detail. Finally, we give advice on how to measure Java performance in practise in Section 4.6 and conclude.

4.2 Statistics

We advocate statistically rigorous data analysis as an important part of a Java performance evaluation methodology. This section describes fundamental statistics theory as described in many statistics textbooks, see for example [62, 71, 82], and discusses how statistics theory applies to Java performance data analysis. The next section then discusses how to add statistical rigour to experiments.

4.2.1 Errors in experimental measurements

As a first step, it is useful to classify errors in two main groups: *systematic errors* and *random errors*. Systematic errors are typically due to some experimental mistake or incorrect procedure which introduces a bias into the measurements. These errors obviously affect the accuracy of the results. It is up to the experimenter to control and eliminate systematic errors. If not, the overall conclusions, even with a statistically rigorous data analysis, may be misleading.

Random errors, on the other hand, are unpredictable and non-deterministic. They are unbiased in that a random error may decrease or increase a measurement. There may be many sources of random errors in the system. In practise, an important concern is the presence of perturbing events that are unrelated to what the experimenter is aiming at measuring, such as external system events, which cause outliers to appear in the measurements. Outliers need to be examined closely, and if the outliers are a result of a perturbing event, they should be discarded. Taking the best measurement also alleviates the issue with outliers, however, we advocate discarding outliers and applying statistically rigorous data analysis to the remaining measurements.

While it is impossible to predict random errors, it is possible to develop a statistical model to describe the overall effect of random errors on the experimental results, which we do next.

4.2.2 Confidence intervals for the mean

In each experiment, a number of samples is taken from an underlying population. A confidence interval for the mean derived from these samples then quantifies the range of values that have a given probability of including the actual population mean. While the way in which a confidence interval is computed is essentially similar for all experiments, a distinction needs to be made depending on the number of samples gathered from the underlying population [35, 71]: (i) the number of samples n is large (typically, $n \geq 30$), and (ii) the number of samples n is small (typically, $n < 30$). We now discuss both cases.

When the number of measurements is large ($n \geq 30$)

Building a confidence interval requires that we have a number of measurements $x_i, 1 \leq i \leq n$, from a population with expected value μ and variance σ^2. The sample mean of these measurements $\overline{x}$ is computed as

$$\overline{x} = \frac{1}{n} \sum_{i=1}^{n} x_i. \tag{4.1}$$

We will approximate the actual true value μ by the mean of our measurements $\overline{x}$ and we will compute a range of values $[c_1, c_2]$ around $\overline{x}$ that defines the confidence interval at a given probability (called the confidence level). The *confidence interval* $[c_1, c_2]$ is defined such that the probability c_1 and c_2 enclosing μ equals $1 - \alpha$; α is called the *significance level* and $(1 - \alpha)$ is called the *confidence level*.

Computing the confidence interval builds on the central limit theory. This theory states that, for large values of n (typically $n \geq 30$), $\overline{x}$ is approximately normally distributed with expected value μ and standard deviation $\sigma/\sqrt{n}$, provided that the samples $x_i, 1 \leq i \leq n$, are (i) independent and (ii) come from the same population with expected value μ and finite standard deviation σ.

Because the significance level α is chosen a priori, we need to determine c_1 and c_2 such that $P\{c_1 \leq \mu \leq c_2\} = 1 - \alpha$ holds. Typically, c_1 and c_2 are chosen to form a symmetric interval around $\overline{x}$, i.e., $P\{\mu < c_1\} = P\{\mu > c_2\} = \alpha/2$. Applying the central-limit theorem, we find that

$$\begin{cases} c_1 & = \overline{x} - z_{1-\frac{\alpha}{2}} \frac{s}{\sqrt{n}} \\ c_2 & = \overline{x} + z_{1-\frac{\alpha}{2}} \frac{s}{\sqrt{n}} \end{cases}, \tag{4.2}$$

with $\overline{x}$ the sample mean, n the number of measurements and s the sample standard deviation computed as follows:

$$s = \sqrt{\frac{1}{n-1} \sum_{i=1}^{n} (x_i - \overline{x})^2}.$$

The value $z_{1-\alpha/2}$ is defined such that a random variable Z that is normally distributed with mean $\mu = 0$ and variance $\sigma^2 = 1$, obeys the following property:

$$P\{Z \leq z_{1-\frac{\alpha}{2}}\} = 1 - \frac{\alpha}{2},$$

i.e., it is the inverse cumulative probability of a standard normal distribution at $1 - \frac{\alpha}{2}$. The value $z_{1-\frac{\alpha}{2}}$ is typically obtained from a precomputed table.

When the number of measurements is small ($n < 30$)

A basic assumption made in the above derivation is that the sample variance s^2 provides a good estimate of the actual variance σ^2. This assumption enabled us to approximate $z = (\overline{x} - \mu)/(\sigma/\sqrt{n})$ as a standard normally distributed random variable, and by consequence to compute the confidence interval for $\overline{x}$. This is generally the case for experiments with a large number of samples, e.g., $n \geq 30$.

However, for a relatively small number of samples, which is typically assumed to be $n < 30$, the sample variance s^2 can be significantly different from the actual variance σ^2 of the underlying population [71]. In this case, it can be

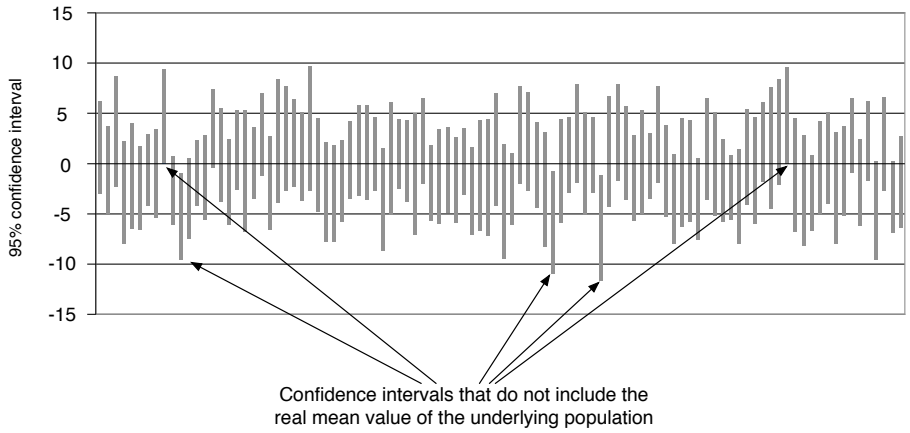


Figure 4.1: An example illustrating the meaning of a 95% confidence interval. Each of the 100 vertical lines indicates an interval; the mean value of the underlying population is 0. In 5 cases the confidence interval does not contain the true mean value.

shown that the distribution of the transformed value $t = (\overline{x} - \mu)/(s/\sqrt{n})$ follows the *Student* t-distribution with $n-1$ degrees of freedom. The confidence interval can then be computed as:

$$\begin{cases} c_1 & = \overline{x} - t_{1-\frac{\alpha}{2};n-1}\frac{s}{\sqrt{n}} \\ c_2 & = \overline{x} + t_{1-\frac{\alpha}{2};n-1}\frac{s}{\sqrt{n}} \end{cases}, \tag{4.3}$$

with the value $t_{1-\alpha/2;n-1}$ defined such that a random variable T that follows the *Student* t distribution with $n-1$ degrees of freedom, obeys:

$$P\{T \leq t_{1-\frac{\alpha}{2};n-1}\} = 1 - \frac{\alpha}{2}.$$

The value $t_{1-\alpha/2;n-1}$ is typically obtained from a precomputed table. It is interesting to note that as the number of measurements n increases, the *Student* t-distribution approaches the normal distribution.

Remark 4.1. *In order to interpret experimental results with confidence intervals, we need to have a good understanding of what a confidence interval actually means. A 90 % confidence interval, i.e., a confidence interval with a 90 % confidence level, means that there is a 90 % probability that the actual distribution mean of the underlying population, μ, is within the confidence interval. Increasing the confidence level to 95% means that we are increasing the probability that the actual mean is within the confidence interval. Since we do not change our measurements, the only way to increase the probability of the mean being within this new confidence interval is to increase its size. By consequence, a 95 % confidence interval will be larger than a 90 % confidence interval; likewise, a 99 % confidence interval will be larger than a 95 % confidence interval.*

This is illustrated in Figure 4.1. The figure shows 100 confidence intervals with a 95 % confidence level for a given population with mean value 0. Each of these intervals was computed based on a random sample of size 50 taken from the population. It can readily be observed that in 5 cases the confidence interval does no contain the true population mean.

Remark 4.2. *It is also important to emphasise that computing confidence intervals does not require that the underlying data is normally distributed. The central limit theory, which is at the foundation of the confidence interval computation, states that $\overline{x}$ is normally distributed irrespective of the underlying distribution of the population from which the measurements are taken. In other words, even if the population is not normally distributed, the average measurement mean $\overline{x}$ is approximately normally distributed if the measurements are taken independently from each other.*

4.2.3 Comparing two alternatives

So far, we were only concerned about computing the confidence interval for the mean of a single system. In terms of a Java performance evaluation setup, this is a single Java benchmark with a given input running on a single virtual machine with a given heap size running on a given hardware platform. However, in many practical situations, a researcher or benchmarker wants to compare the performance of two or more systems. In this section, we focus on comparing two alternatives; the next section then discusses comparing more than two alternatives. A practical use case scenario could be to compare the performance of two virtual machines running the same benchmark with a given heap size on a given hardware platform. Another example use case is to compare the performance of two garbage collectors for a given benchmark, heap size and virtual machine on a given hardware platform.

When comparing two alternatives, we wish to find support that allows us to reject the null hypothesis, $H_0 \equiv \mu_1 = \mu_2 \equiv \mu_1 - \mu_2 = 0$, where μ_i is the mean value for the population of alternative i.

The standard test for comparing two alternatives is the following. Consider two alternatives with n_1 measurements for the first alternative and n_2 measurements for the second alternative. We then first determine the sample means $\overline{x}_1$ and $\overline{x}_2$ and the sample standard deviations s_1 and s_2. We subsequently compute the difference of the means as $\overline{x} = \overline{x}_1 - \overline{x}_2$. The standard deviation s_x of this difference of the mean values is then computed as:

$$s_x = \sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}}.$$

The confidence interval for the difference of the means is then given by

$$\begin{cases} c_1 & = \overline{x} - z_{1-\alpha/2} s_x \\ c_2 & = \overline{x} + z_{1-\alpha/2} s_x \end{cases}. \tag{4.4}$$

If this confidence interval includes zero, we can conclude, at the confidence level chosen, that there is no statistically significant difference between the two alternatives, and we do not reject H_0. If, on the other hand $0 \notin [c_1, c_2]$, we reject H_0 at a confidence level of $1 - \alpha$. Note the careful wording here. Even if we reject the null hypothesis, there is still a probability α that the differences observed in our measurements are simply due to random effects in our measurements. In other words, we cannot assure with a 100 % certainty that there is an actual difference between the compared alternatives. In some cases, taking such 'weak' conclusions may not be very satisfying – people tend to like strong and affirmative conclusions – but it is the best we can do given the statistical nature of the measurements.

Equation 4.4 only holds in case the number of measurements is large on both systems, i.e., $n_1 \geq 30$ and $n_2 \geq 30$. In case the number of measurements on at least one of the two systems is smaller than 30, then we can no longer assume that the difference of the means is normally distributed. We then need to resort to the *Student* t-distribution. The test statistic $t = (\mu_1 - \mu_2)/s_x$ is then distributed as a Student t-distribution with n_{df} degrees of freedom. Consequently, we obtain the correct confidence interval $[c_1, c_2]$ by replacing the value $z_{1-\alpha/2}$ in the above formula with $t_{1-\alpha/2;n_{df}}$; the degrees of freedom n_{df} is then to be approximated by the integer number nearest to

$$n_{df} = \left(\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}\right)^2 \Big/ \frac{(s_1^2/n_1)^2}{n_1 - 1} + \frac{(s_2^2/n_2)^2}{n_2 - 1}.$$

Associated with a statistical test is the so-called p-value. This value indicates the probability that the test statistic used has at least the same significance as it would have under the assumption that the null hypothesis H_0 holds. Consequently, the smaller the p-value, the more likely that H_0 is false, i.e., the evidence against H_0 is stronger. Therefore, if the p-value is smaller than the significance level of the test, we reject H_0 in favour of the alternative hypothesis H_a[1].

Because a visual representation can be quite appealing to people, they often resort to another approach to compare two alternatives. Perhaps the simplest – and often misused – visual approach to comparing two alternatives is to determine whether the confidence intervals for the two alternatives overlap; we call this the *overlap-method*. If the confidence intervals at a $1 - \alpha$ confidence level for the alternatives do *not* overlap, then there is a statistically significant difference at the α significance level [95], i.e., the null hypothesis is also rejected by a more rigorous test (e.g., a *Student* t-test) at the same significance level. The converse however is not true: if the overlap-method does not reject the null hypothesis, it does not mean there is no statistically significant difference. Hence, looking at the overlap is thus more conservative. Moreover, the overlap-method is less powerful, because it more often fails to reject a false null hypothesis compared to a standard test, such as described above [95]. As

[1] Note that we never *accept* H_a, we simply reject H_0.

such, it is advisable to not place to much trust in this approach, and it certainly should not be used for formal significance testing[2].

4.2.4 Confidence intervals for speedup ratios

For the sake of completeness, we also provide the correct equations to compute a confidence interval for speedup, i.e., the relative performance of one workload versus a second workload. One might be tempted to simply compute confidence intervals $[c_1^1, c_2^1]$ and $[c_2^1, c_2^2]$ for each workload and derive a new confidence interval as follows:

$$\left[\frac{c_1^x}{c_2^y}, \frac{c_2^x}{c_1^y}\right]$$

This, however, would result in a confidence interval width that is larger than the confidence interval width for either workload. What is required is an estimator for the ratio of both workloads' performance. This can be achieved using the sample data as follows [35, 73].

Given the samples for the first workload as $\{x_1, \ldots, x_n\}$, and for the second workload as $\{y_1, \ldots, y_n\}$, the estimator $\hat{R}$ for the real ratio of both populations R is computed by

$$\hat{R} = \frac{\overline{y}}{\overline{x}} = \frac{\sum_{i=1}^{n} y_i}{\sum_{i=1}^{n} x_i}$$

If the population size N is much larger than the number of samples n , the variance for R is estimated as

$$v = \frac{1 - \frac{n}{N}}{n\overline{x}^2}\left(s_y^2 + \hat{R}^2 s_x^2 - 2\hat{R}s_{xy}\right)$$

with

$$s_{xy} = \frac{\sum_{i=1}^{n}(x_i - \overline{x})(y_i - \overline{y})}{n-1}.$$

Finally, if n is large the confidence interval is given by

$$\begin{cases} c_1 & = \hat{R} - z_{1-\frac{\alpha}{2}}\sqrt{v} \\ c_2 & = \hat{R} + z_{1-\frac{\alpha}{2}}\sqrt{v} \end{cases},$$

If n is small, the z-statistic should be replaced by the *Student* t-statistic.

4.2.5 Comparing more than two alternatives: ANOVA

The approach discussed in the previous section to comparing two alternatives is simple and intuitively appealing, however, it is limited to comparing two alternatives. ANOVA is more general and more robust technique for comparing

[2]For example, even in the case of comparisons between groups with an equal sample size and sample variance, the degree of overlap between 95 % confidence intervals can be as high as 29 % before we will reject the null hypothesis with a p-value of 0.05 [6].

multiple alternatives. As we briefly mentioned in Chapter 2, ANOVA [62, 82] separates the total variation in a set of measurements into a component due to random fluctuations in the measurements and a component due to the actual differences among the alternatives. In other words, ANOVA separates the total variation observed in (i) the variation observed *within* each alternative, which is assumed to be a result of random effects in the measurements, and (ii) the variation *between* the alternatives. If the variation between the alternatives is larger than the variation within each alternative, then it can be concluded that there is a statistically significant difference between the alternatives. ANOVA assumes that the variance in measurement error is the same for all of the alternatives. Also, ANOVA assumes that the errors in the measurements for the different alternatives are independent and normally distributed. However, ANOVA is fairly robust towards non-normality, especially in case there is a balanced number of measurements for each of the alternatives.

To present the general idea behind ANOVA, it is convenient to organize the measurements as shown in Table 4.1: there are $n \cdot k$ measurements – n measurements for all k alternatives. The column means are defined as:

$$\overline{y}_{.j} = \frac{\sum_{i=1}^{n} y_{ij}}{n},$$

and the overall mean is defined as:

$$\overline{y}_{..} = \frac{\sum_{j=1}^{k} \sum_{i=1}^{n} y_{ij}}{n \cdot k}.$$

It is then useful to compute the variation due to the effects of the alternatives, sum-of-squares due to the alternatives (SSA), as the sum of the squares of the differences between the mean of the measurements for each alternative and the overall mean, or:

$$SSA = n \sum_{j=1}^{k} (\overline{y}_{.j} - \overline{y}_{..})^2. \tag{4.5}$$

The variation due to random effects within an alternative is computed as the sum of the squares of the differences (or errors) (SSE) between the individual measurements and their respective alternative mean, or:

$$SSE = \sum_{j=1}^{k} \sum_{i=1}^{n} (y_{ij} - \overline{y}_{.j})^2. \tag{4.6}$$

Finally, the sum-of-squares total, SST, or the sum of squares of the differences between the individual measurements and the overall mean is defined as:

$$SST = \sum_{j=1}^{k} \sum_{i=1}^{n} (y_{ij} - \overline{y}_{..})^2. \tag{4.7}$$

It can be shown that

$$SST = SSA + SSE.$$

Measurements	Alternatives						Overall mean
	1	2	...	j	...	k	
1	y_{11}	y_{12}	...	y_{1j}	...	y_{1k}	
2	y_{21}	y_{22}	...	y_{2j}	...	y_{2k}	
⋮	⋮	⋮	⋱	⋮	⋱	⋮	
i	y_{i1}	y_{i2}	...	y_{ij}	...	y_{ik}	
⋮	⋮	⋮	⋱	⋮	⋱	⋮	
n	y_{n1}	y_{n2}	...	y_{nj}	...	y_{nk}	
Column means	$\overline{y}_{.1}$	$\overline{y}_{.2}$	...	$\overline{y}_{.j}$	...	$\overline{y}_{.k}$	$\overline{y}_{..}$

Table 4.1: Organising the n measurements for k alternatives in a single-factor ANOVA analysis.

Or, in other words, the total variation can be split up into a *within* alternative (SSE) component and a *between* alternatives (SSA) component.

The intuitive understanding of an ANOVA analysis now is to quantify whether the variation across alternatives, SSA, is 'larger' in some statistical sense than the variation within each alternative, SSE, which is due to random measurement errors. A simple way is to compare the fractions SSA/SST versus SSE/SST. A statistically more rigorous approach is to apply a statistical test, called the *F-test*, which is used to test whether two variances are significantly different [71].

The *F statistic* is computed as

$$F = \frac{s_a^2}{s_e^2}, \tag{4.8}$$

with

$$s_a^2 = \frac{SSA}{k-1}, \text{ and } s_e^2 = \frac{SSE}{k(n-1)}; \tag{4.9}$$

with k the number of alternatives, and n the number of measurements per alternative. If this F statistic is larger than the critical value $F_{[1-\alpha;(k-1),k(n-1)]}$, which is to be obtained from a precomputed table, we can say that the variation due to differences among the alternatives is significantly larger than the variation due to random measurement noise, at the α level of significance.

Simultaneous pairwise confidence intervals

After completing an ANOVA test, we may conclude that there is a statistically significant difference between the alternatives. However, the ANOVA test does not tell us between which alternatives there is a statistically significant difference. There exist a number of techniques to find out between which alternatives there is or there is not a statistically significant difference. One approach is the Tukey HSD (Honestly Significantly Different) [82] test. The

advantage of the Tukey HSD test over simpler approaches, such as pairwise Student t-tests for comparing means, is that it limits the probability of making an incorrect conclusion in case there is no statistically significant difference between all the means and in case most of the means are equal but one or two are different. More concretely, the Tukey HSD test will yield a set of pairwise comparisons with the collective desired confidence level.

Given that we want to compare all k factor alternatives using a pairwise test, we obtain a family of tests with

$$H_0 = \mu_i - \mu_j = 0$$
$$H_a = \mu_i - \mu_j \neq 0$$

where $i \in \{1, \ldots, k\}$, $j \in \{1, \ldots, k\}$ and $i \neq j$. In this case, the point estimator for the difference between μ_i and μ_j is given by the difference $\hat{D} = \overline{y}_{.i} - \overline{y}.j$. It can be shown that the test statistic

$$Q = \sqrt{2}\hat{D}/s_D$$

is distributed as a studentized range q, for which the values can be obtained from a precomputed table. s_D is given by

$$s_D = s_e^2(1/n_i + 1/n_j).$$

If $|Q| > Q_{1-\alpha;k;nk-k}$ we reject H_0 in favour of H_a. For a more detailed discussion, we refer to specialised literature [82].

In summary, an ANOVA analysis allows for varying one *input variable* within the experiment. For example, in case a benchmarker wants to compare the performance of four virtual machines for a given benchmark, a given heap size and a given hardware platform, the virtual machine then is the input variable and the four virtual machines are the four alternatives. Another example where an ANOVA analysis can be used is when a benchmarker wants to compare the performance of various garbage collectors for a given virtual machine, a given benchmark and a given system setup.

4.2.6 Multi-factor and multivariate experiments

Multi-factor ANOVA. The ANOVA analysis discussed in the previous section is a so called one-factor ANOVA, meaning that only a single input variable can be varied during the setup – the values this input variable can take are called *factor levels*. A multi-factor ANOVA allows for studying the effect of multiple input variables and all of their interactions, along with an indication of the magnitude of the measurement error. For example, an experiment where both the garbage collector and the heap size are varied, could provide deep insight into the effect on overall performance of both the garbage collector and the heap size individually as well as the interaction of both the garbage collector and the heap size.

Multivariate ANOVA. The ANOVA analyses discussed so far only consider, what is called, a single dependent variable. In a Java context, this means that an ANOVA analysis only allows for making conclusions about a single performance metric, e.g., execution time. However, a benchmarker might be interested in more characteristics than execution time. For example, he might wish to consider other metrics such as energy consumption or other non-functional metrics. A multivariate ANOVA or MANOVA allows for considering multiple dependent variables within one single experiment. The key point of performing a MANOVA instead of multiple ANOVA analyses on the individual dependent variables, is that a MANOVA analysis takes into account the correlation across the dependent variables whereas multiple ANOVAs do not.

4.2.7 Discussion

In the previous sections, we explored a wide range of statistical techniques and we discussed how to apply these techniques within a Java performance evaluation context. However, using the more complex analyses, such as multi-factor ANOVA and MANOVA, raises two concerns. First, their output is often non-intuitive and in many cases hard to understand without deep background knowledge in statistics – this might go beyond what can be reasonably expected from a software developer doing the benchmarking experiment. Second, as mentioned before, doing all the measurements required as input to the analyses can be very time-consuming, up to the point where it becomes intractable. For these reasons, we limit ourselves to a Java performance evaluation methodology that is practical yet statistically rigorous. The methodology that we present computes confidence intervals which allows for doing comparisons between alternatives on a per-benchmark basis, as discussed in Sections 4.2.3 and 4.2.5. Of course, a benchmarker who is knowledgeable in statistics may perform more complex analyses.

As a summary, Figure 4.2 shows a decision tree to decide which technique to use. The first choice is deciding if a multivariate or a univariate analysis should be used. In the former case, we still recommend performing a univariate analysis on each separate characteristic, should the MANOVA reject the null hypothesis of equal mean performance across all factors. Similarly, in the multivariate case with but two factor levels for a single factor – for example, comparing the cache events (characteristics) on a single virtual machine between two versions of an application – we still advise to compare each characteristic separately using a *Student t*-test, should the Hotelling test[3] show there is a significant difference between the two factor levels. In the univariate case, when either the number of factors is larger than 1, or the number of factor levels is larger than 2, we recommend performing an ANOVA. Of course, the form of the null hypothesis will be different in both cases. The complete specification is beyond the scope of this dissertation, but can be found in any

[3]The Hotelling test is the multivariate form of the *Student t*-test.

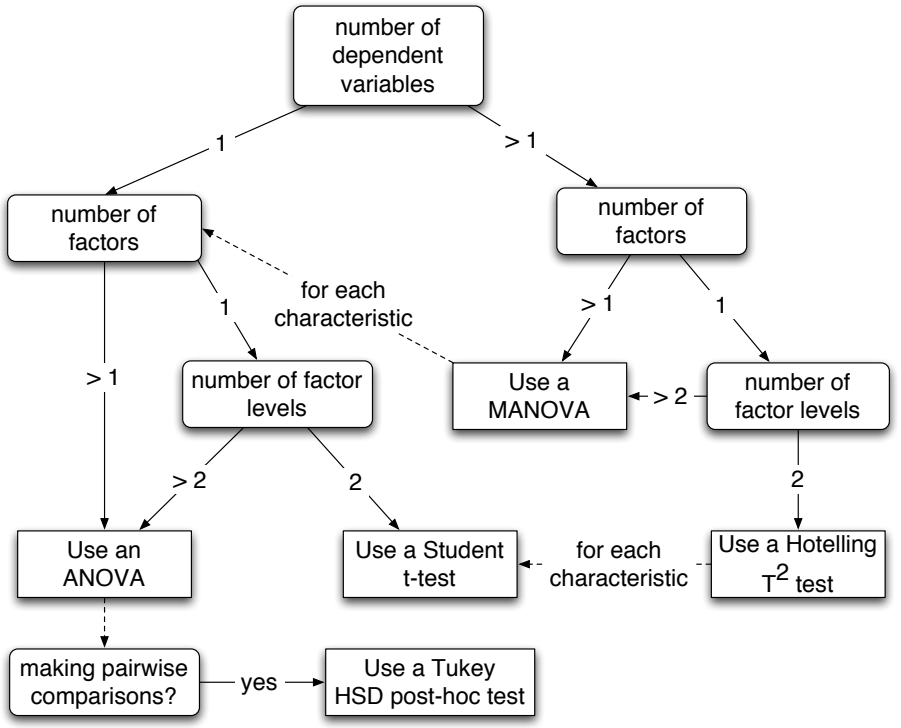


Figure 4.2: Decision tree to determine which statistical technique should be used for analysing Java performance.

textbook on ANOVA [82]. Only when there are but two factor levels for a single factor do we recommend using a *Student t*-test. Finally, when multiple simultaneous pairwise comparisons are desired, it is prudent to use a post-hoc test, such as the Tukey HSD test.

4.3 Statistically rigorous data analysis

Having discussed the general theory of statistics and how it relates to Java performance evaluation, we now suggest statistically rigorous methodologies for quantifying start-up and steady-state Java performance by combining a number of existing approaches. The evaluation section in this chapter then compares the accuracy of prevalent data analysis methodologies against these statistically rigorous methodologies.

Notation. We refer to x_{ij} as the measurement of the j-th benchmark iteration of the i-th VM invocation.

4.3.1 Start-up performance

The goal of measuring start-up performance is to measure how quickly a Java virtual machine can execute a relatively short-running Java program.

For measuring start-up performance, we advocate a two-step methodology:

1. Measure the execution time of $p > 1$ VM invocations, each VM invocation running a single benchmark iteration. This results in p measurements x_{ij} with $1 \leq i \leq p$ and $j = 1$.
2. Compute the confidence interval for a given confidence level as described in Section 4.2.2. If there are more than 30 measurements, use the standard normal z-statistic; otherwise use the *Student* t-statistic.

The methodology is also illustrated in Figure 4.3, where only the first iteration is used in each virtual machine invocation.

Recall that the central limit theory assumes that the measurements are independent. This may not be true in practise, because the first VM invocation in a series of measurements may change system state that persists past this first VM invocation, such as dynamically loaded libraries persisting in physical memory or data persisting in the disk cache. To reach a level of independence in the measurements, we discard the first VM invocation for each benchmark from our measurements and only retain the subsequent measurements, as done by several other researchers; this ensures the libraries are already loaded when doing the measurements.

4.3.2 Steady-state performance

The goal of measuring steady-state performance is to quantify the effect of the various optimisations performed by the virtual machine on a longer running application.

There are two issues with quantifying steady-state performance. The first issue is to determine when steady-state performance is reached. Long-running applications typically run on large or streaming input data sets. Benchmarkers typically approximate long-running benchmarks by running existing benchmarks with short inputs multiple times within a single VM invocation, i.e., the benchmark is iterated multiple times. In a real-life situation, many applications are run inside a *server* virtual machine over and over without restarting the virtual machine. Iterating a benchmark mimics this behaviour. The question then is how many benchmark iterations do we need to consider before we reach steady-state performance within a single

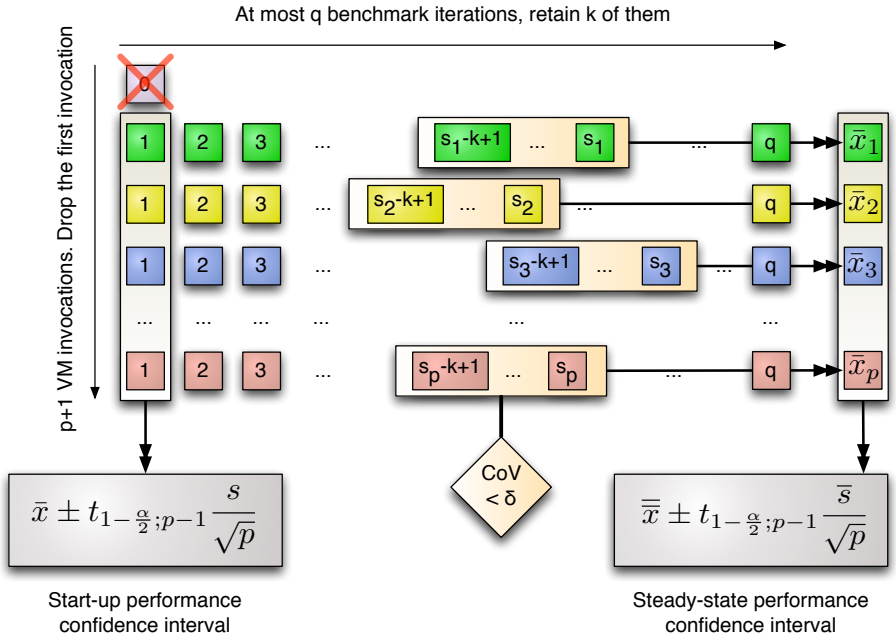


Figure 4.3: Scheme to rigorously determine both start-up and steady-state performance. The algorithm for steady-state has five parameters: the (maximum) number of invocations p, the maximum number of iterations q, the number of retained iterations k, the CoV threshold δ and the significance level α. For start-up, only two of these parameters are used: p and α. In steady-state regime for the i-th VM invocation, we stop the execution after iteration s_i. For all i, the inequality $s_i \leq q$ holds. Furthermore, not all s_i have the same value. $\overline{x}$ is the mean of the first iterations across all invocations. $\overline{\overline{x}}$ is the mean of the $\overline{x}_i, i \in \{1, \ldots, p\}$. s is the standard deviation of the first iterations in each invocation and $\overline{s}$ is the standard deviation of the aforementioned $\overline{x}_i$.

VM invocation? This is a difficult question to answer in general; the answer will differ from application to application, and in some cases it may take a very long time before steady-state is reached.

The second issue with steady-state performance is that different VM invocations running multiple benchmark iterations may result in different steady-state performances [5]. Different methods may be optimised at different levels of optimisation across different VM invocations, changing steady-state performance.

To address these two issues, we advocate a four-step methodology for quantifying steady-state performance, see also Figure 4.3:

1. Consider p VM invocations, each VM invocation running at most q benchmark iterations. Suppose that we want to retain k measurements per invocation.

2. For each VM invocation i, determine the iteration s_i where steady-state performance is reached, i.e., once the coefficient of variation (CoV) of the k iterations ($s_i - k$ to s_i) falls below a preset threshold, say 0.01 or 0.02.

3. For each VM invocation, compute the mean $\overline{x}_i$ of the k benchmark iterations under steady-state:

$$\overline{x}_i = \sum_{j=s_i-k}^{s_i} x_{ij}.$$

4. Compute the confidence interval for a given confidence level across the computed means from the different VM invocations. The overall mean equals $\overline{\overline{x}} = \sum_{i=1}^{p} \overline{x}_i$, and the confidence interval is computed over the $\overline{x}_i$ measurements.

We thus first compute the mean $\overline{x}_i$ across multiple iterations within a single VM invocation i, and subsequently compute the confidence interval across the p VM invocations using the $\overline{x}_i$ means, see steps 3 and 4 from above. The reason for doing so is to reach independence across the measurements from which we compute the confidence interval: the various iterations within a single VM invocation are not independent, however, the mean values $\overline{x}_i$ across multiple VM invocations are assumed to be independent.

4.4 Experimental setup

To demonstrate the existence of a pitfall in current Java performance analysis, we will set up an experiment in which we compare five garbage collection algorithms. We compare the accuracy of prevalent data analysis techniques against a statistically rigorous approach. Before doing so, we first describe our experimental environment. We use the benchmarks from SPECjvm98 and DaCapo, listed in Table 4.2.

We use a single virtual machine, namely the Jikes RVM. We use the SVN version from February 12, 2007, built according to the FastAdaptive configuration. The MMTk [16] that accompanies Jikes RVM offers several garbage collection strategies. Of these, we use five strategies for which all benchmarks from Table 4.2 run to completion: (1) CopyMS, (2) GenCopy, (3) GenMS, (4) MarkSweep, and (5) SemiSpace. We did not use RefCount because the execution time for this strategy is much higher and significantly worse than for any other strategy. We did not use MarkCompact or GenRC because we were unable to run Jikes RVM with these collectors for several benchmarks.

Each benchmark is able to complete a run successfully for a certain heap size: the minimal heap size for that particular benchmark. In our experiments we consider a per-benchmark heap size range, following [15]. We vary the

Benchmark	Description	Minimal heap size (MB)
compress	file compression	24
jess	puzzle solving	32
db	database	32
javac	Java compiler	32
mpegaudio	MPEG decompression	16
mtrt	raytracing	32
jack	parsing	24
antlr	parsing	32
bloat	Java bytecode optimization	56
fop	PDF generation from XSL-FO	56
hsqldb	database	176
jython	Python interpreter	72
luindex	document indexing	32
pmd	Java class analysis	64

Table 4.2: The SPECjvm98 (top seven) and DaCapo (bottom seven) benchmarks considered in this experiment. The rightmost column indicates the minimum heap size, as a multiple of 8 MB, for which all GC strategies run to completion.

heap size from its minimal size up to six times that amount, in increments of $1/4$ the minimal size.

Following the advice by Blackburn et al. [17], we consider multiple hardware platforms in our performance evaluation methodology: a 2.1GHz AMD Athlon XP, a 2.8GHz Intel Pentium 4, and a 1.42GHz PowerPC G4 machine. The AMD Athlon and Intel Pentium 4 have 2GB of main memory; the PowerPC G4 has 1GB of main memory. These machines run the Linux operating system, version 2.6.18. In all of our experiments we consider an otherwise idle and unloaded machine.

4.5 Evaluating prevalent methodologies

In our experiments, we compare the overall performance of the five garbage collection strategies mentioned in the previous section for each of the heap sizes mentioned above. In this experiment, we expect to encounter both large and small differences between the alternative garbage collection strategies, making this a good use case for demonstrating our approach. This experiment is similar to what is being done in GC literature. In fact, these five GC strategies pose a complex space-time trade-off, and it is unclear which strategy is the winner without doing a detailed experiment.

Computing confidence intervals for the statistically rigorous methodology

Statistically rigorous methodology $H_0 \equiv \mu = \mu_A - \mu_B = 0$	Performance difference for the prevalent methodology $\lvert a - b \rvert \leq \theta$	$\lvert a - b \rvert > \theta$
H_0 is not rejected	*indicative*	*misleading*
H_0 is rejected, but the ordering of the prevalent methodology is maintained	*misleading but correct*	*correct*
H_0 is rejected, and the ordering of the prevalent methodology is reversed	*misleading and incorrect*	*incorrect*

Table 4.3: Classifying conclusions by a prevalent methodology in comparison to a statistically rigorous methodology.

is done, following Section 4.2, by applying an ANOVA and a Tukey HSD test to compute simultaneous 95 % confidence intervals for all the GC strategies per benchmark and per heap size, i.e., the factor levels (the alternatives) in the ANOVA are given by the five GC strategies.

To evaluate the accuracy of the prevalent performance evaluation methodologies we consider all possible pairwise GC strategy comparisons for all heap sizes. For each heap size, we then determine whether prevalent data analysis leads to the same conclusion – which is the faster alternative – as statistically rigorous data analysis. There are $C_5^2 = 10$ pairwise GC comparisons per heap size and per benchmark. Or, 210 GC comparisons in total across all heap sizes per benchmark.

We now classify all of these comparisons in six categories, see Table 4.3, and then report the relative frequency of each of these six categories. These results help us better understand the frequency of misleading and incorrect conclusions using prevalent performance methodologies. In each experiment, the null hypothesis is that the strategies under consideration, A and B, have equal mean execution times, i.e., $H_0 \equiv \mu = \mu_A - \mu_B = 0$. We make a distinction between (i) confidence intervals for μ that contain 0, i.e., H_0 is not rejected, and (ii) confidence intervals for μ that do not contain 0, i.e., H_0 is rejected with a significance level α. In what follows, we denote the performance number given by the prevalent approach as a and b for the alternatives A and B, respectively.

H_0 is *not* rejected. This indicates that the performance differences observed may be due to random fluctuations. As a result, any conclusion taken by a methodology that concludes that one alternative performs better than another is questionable. The only valid conclusion in this case is that there is no statistically significant difference between the alternatives.

Benchmarkers using prevalent performance analysis typically do not state that one alternative is better than another when the performance difference is very small though. To mimic this practise, we introduce a threshold θ to classify decisions: a performance difference smaller than θ is considered a small performance difference and a performance difference larger than θ is considered a large performance difference. We will vary the θ threshold from 1 % up to 3 % in our evaluation.

Now, in case the performance difference by the prevalent methodology is considered large, we conclude the prevalent methodology to be *misleading*. In other words, the prevalent methodology says there is a significant performance difference whereas the statistically rigorous method concludes that this performance difference may be due to random fluctuations. If the performance difference is small based on the prevalent methodology, we consider the prevalent methodology to be *indicative*. This means that both approaches arrive at the same conclusion, yet we want to point out that the prevalent method does provide conclusive statistical evidence for this conclusion.

H_0 is rejected. This means that we can conclude that there is a statistically significant performance difference among the alternatives for a given significance level α. There are two possibilities for the relative positioning of the confidence interval $[c_1, c_2]$ for μ. Either the sign of all $x \in [c_1, c_2]$ equals the sign of $a - b$, and the conclusion made by both approaches is the same. In that case the prevalent methodology is considered *correct*. Or, the sign of all $x \in [c_1, c_2]$ is the opposite of the sign of $a - b$, in which case the methodology is considered *incorrect*, because both approaches yield contradictory conclusions.

To incorporate a performance analyst's subjective judgement, modelled through the θ threshold from above, we make one more distinction based on whether the performance difference is considered small or large. In particular, if the prevalent methodology states there is a small difference, the conclusion is classified to be *misleading*. In fact, there is a statistically significant performance difference, however, the performance difference is small. This means that the prevalent approach would conclude that the two alternatives are not different, whereas the rigorous approach does find a statistically significant – albeit small – difference.

Consequently, we have four classification categories for non-overlapping confidence intervals[4], see Table 4.3. If the performance difference by the prevalent methodology is larger than θ, and the ranking by the prevalent methodology equals the ranking by the statistically rigorous methodology, then the prevalent methodology is considered to be *correct*; if the prevalent methodology has the opposite ranking as the statistically rigorous methodology, the prevalent methodology is considered *incorrect*. In case of a small performance

[4]Recall that for non-overlapping intervals, the null hypothesis H_0 will be rejected at the α significance level.

difference according to the prevalent methodology, and the same ranking as the statistically rigorous methodology, the prevalent methodology is considered to be *misleading but correct*; in case of an opposite ranking, the prevalent methodology is considered *misleading and incorrect*.

We now compare the prevalent data analysis techniques discussed in the previous chapter against the statistically rigorous method for both start-up and steady-state performance. First, we consider non-controlled compilation; replay-compilation is considered later on.

4.5.1 Start-up performance

We first focus on start-up performance. Figures 4.4, 4.5, and 4.6 shows the percentage GC comparisons by the prevalent data analysis approaches leading to indicative, misleading and incorrect conclusions for $\theta = 1\%$ and $\theta = 2\%$ thresholds on the Athlon XP, Pentium4, and PowerPC G4 platforms, respectively. The various bars in these graphs show various prevalent methodologies. There are bars for reporting the best, the second best, the worst, the mean and the median performance number; for 3, 5, 10 and 30 VM invocations and a single benchmark iteration – for example, the 'best of 3' means taking the best performance number out of 3 VM invocations. The statistically rigorous methodology that we compare against considers 30 VM invocations and a single benchmark iteration per VM invocation, and considers 95 % confidence intervals.

There are a number of interesting observations to be made from these graphs.

- First of all, prevalent methods can be misleading in a substantial fraction of comparisons between alternatives, i.e., the total fraction misleading comparisons ranges up to 16 %. In other words, in up to 16 % of the comparisons, the prevalent methodology makes too strong a statement saying that one alternative is better than another.

- For a fair number of comparisons, the prevalent methodology can even lead to incorrect conclusions, i.e., the prevalent methodology says one alternative is better (by more than θ percent) than another, whereas the statistically rigorous methodology takes the opposite conclusion based on non-overlapping confidence intervals. For some prevalent methodologies, the fraction of incorrect comparisons can be more than 3 %.

- We also observe that some prevalent methodologies perform better than others. In particular, mean and median are consistently better than best, second best and worst. The accuracy of the mean and median methods seems to improve with the number of measurements, whereas the best, second best and worst methods do not.

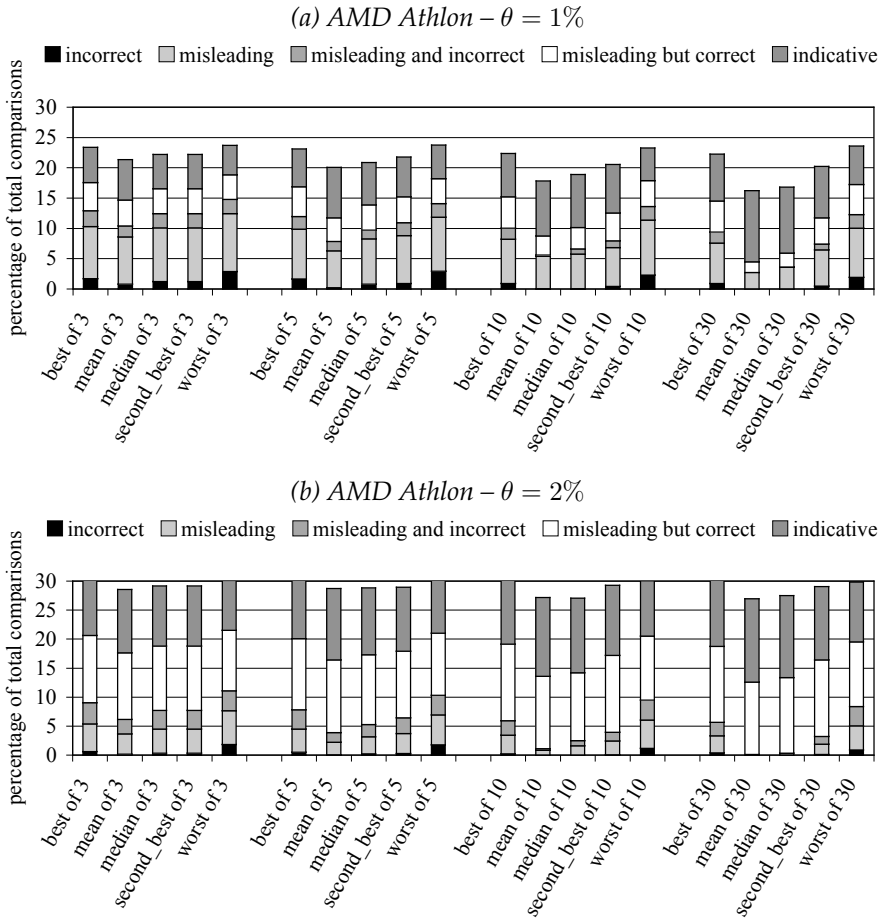


Figure 4.4: Percentage GC comparisons by prevalent data analysis approaches leading to incorrect, misleading or indicative conclusions. Results are shown for the AMD Athlon machine with $\theta = 1\%$ (a) and $\theta = 2\%$ (b).

- Increasing the θ threshold reduces the number of incorrect conclusions by the prevalent methodologies and at the same time also reduces the number of misleading and the number of correct conclusions. By consequence, the number of misleading-but-correct, misleading-and-incorrect and indicative conclusions increases, or, in other words, the conclusiveness of a prevalent methodology reduces with an increasing θ threshold. Figure 4.7 shows the classification as a function of the θ threshold for the *javac* benchmark, which we found to be a representative example benchmark. The important conclusion here is that increasing the θ threshold for a prevalent methodology does not replace a statistically rigorous methodology.

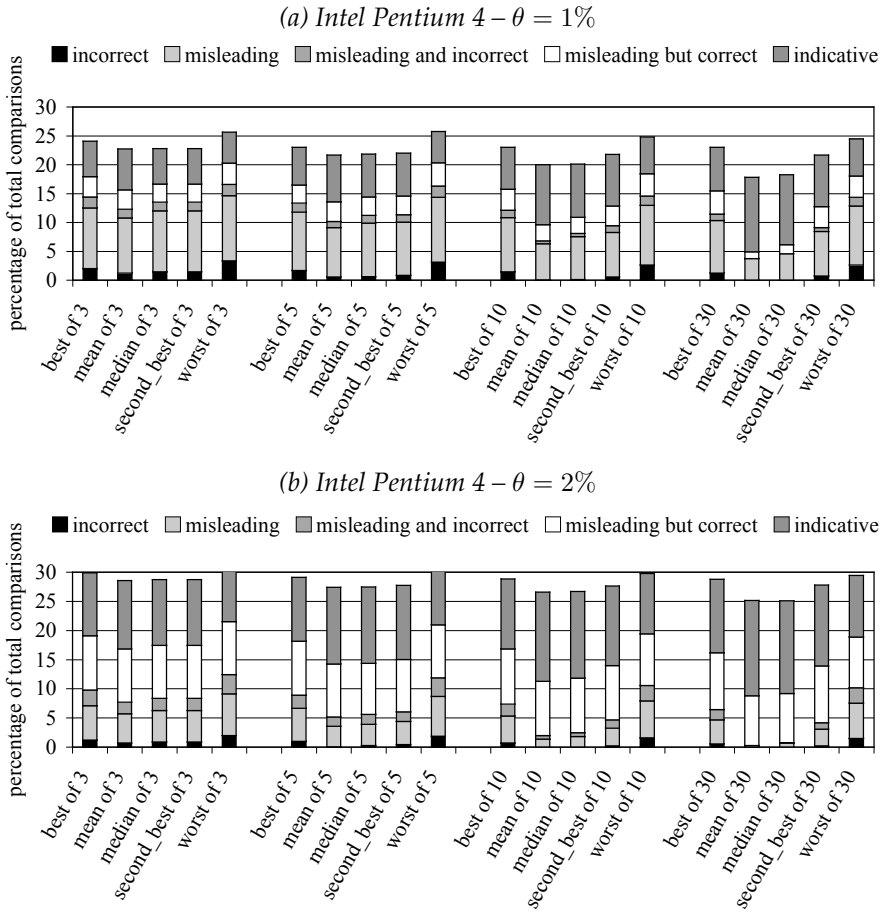


Figure 4.5: Percentage GC comparisons by prevalent data analysis approaches leading to incorrect, misleading or indicative conclusions. Results are shown for the Intel Pentium 4 machine with $\theta = 1\%$ (a) and $\theta = 2\%$ (b).

- One final interesting observation that is consistent with the observations made by Blackburn et al. [17], is that the results presented in Figure 4.4 through Figure 4.6 vary across different hardware platforms. In addition, the results also vary across benchmarks, see Figure 4.8 which shows per-benchmark results for the 'best-of-30' prevalent method; we obtained similar results for the other methods. Some benchmarks are more sensitive to the data analysis method than others. For example, *jess* and *hsqldb* are almost insensitive, whereas other benchmarks have a large fraction misleading and incorrect conclusions; *db* and *javac* for example show more than 3 % incorrect conclusions.

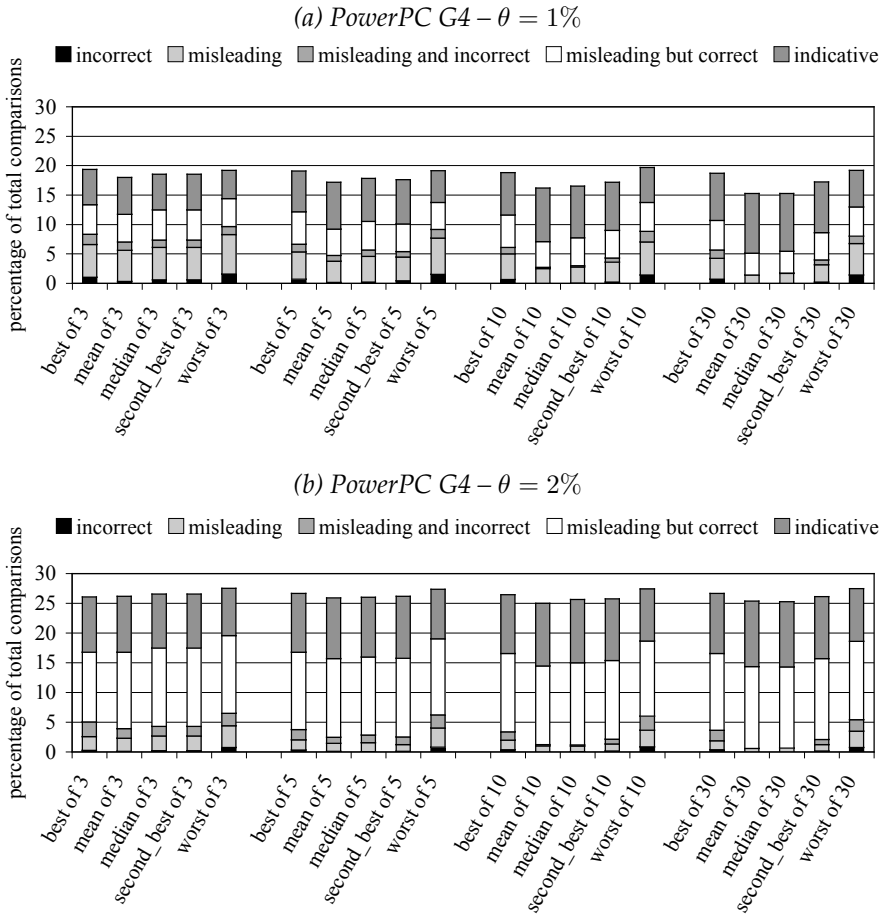


Figure 4.6: Percentage GC comparisons by prevalent data analysis approaches leading to incorrect, misleading or indicative conclusions. Results are shown for the PowerPC G4 machine with $\theta = 1\%$ (a) and $\theta = 2\%$ (b).

A VM developer use case.

The evaluation so far quantified comparing *all* GC strategies against all other GC strategies, a special use case. Typically, a researcher or developer is merely interested in comparing a new GC algorithm against already existing algorithms. To mimic this use case, we compare *one* GC strategy, GenMS, against all other four GC strategies. The results are shown in Figure 4.9 and are very much in line with the results presented in Figure 4.4: prevalent data analysis methods are misleading in many cases, and in some cases even incorrect.

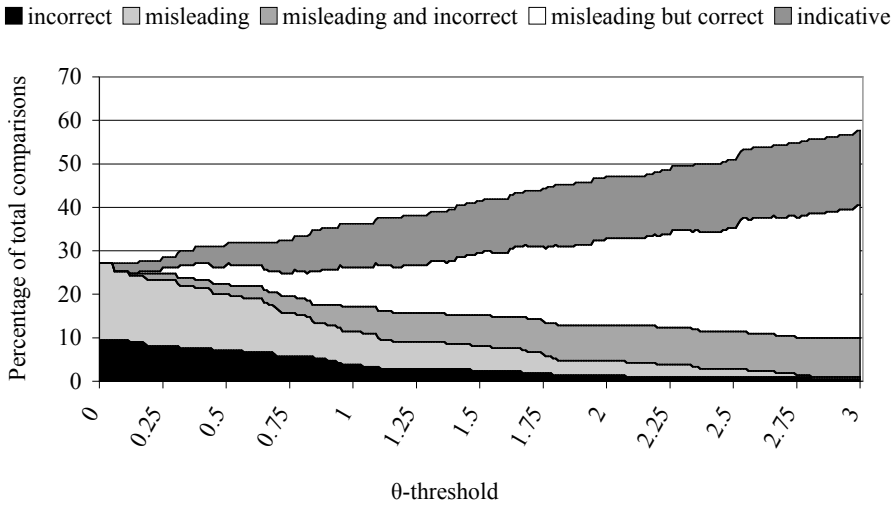


Figure 4.7: The classification for *javac* as a function of the threshold $\theta \in [0; 3]$ for the 'best-of-30' prevalent method, on the AMD Athlon.

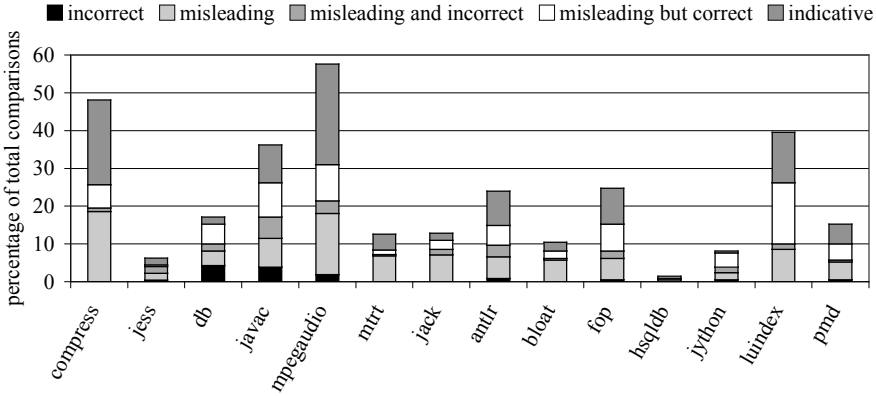


Figure 4.8: Per-benchmark percentage GC comparisons by the 'best-of-30' method classified as misleading, incorrect and indicative on the AMD Athlon machine with $\theta = 1$ %.

An application developer use case.

Our next case study takes a look from the perspective of an application developer by looking at the performance of a single benchmark. Figure 4.10 shows two graphs for *db* for the best of 30 and the confidence interval based performance evaluation methods. The different curves represent different garbage collectors. These graphs clearly show that different conclusions may be taken

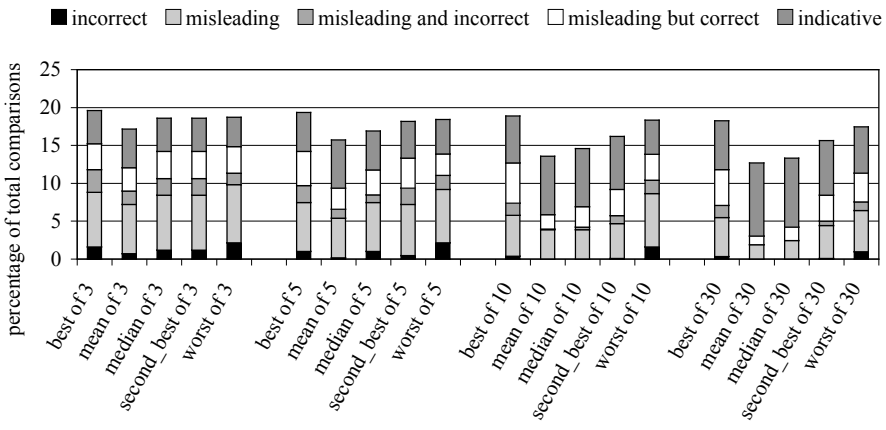


Figure 4.9: The (in)accuracy of comparing the GenMS GC strategy against four other GC strategies using prevalent methodologies, for $\theta = 1$ % on the AMD Athlon machine.

depending on the evaluation method used. For example, for heap sizes between 80 MB and 120 MB, one would conclude using the 'best' method that CopyMS clearly outperforms MarkSweep and performs almost equally well as GenCopy. However, the confidence intervals show that the performance difference between CopyMS and MarkSweep could be due to random fluctuations, and, in addition, the statistically rigorous method clearly shows that GenCopy substantially outperforms CopyMS.

Varying the significance level α

In the previous sections, we have varied value of the threshold θ, mimicking the prevalent practice of defining a point where performance improvements are considered to be noteworthy. We have always used a significance level $\alpha = 0.05$, which corresponds to a confidence level of 95%. The question can be raised if α has a major impact on our conclusions. In Figure 4.11, we show the percentage of comparisons leading to each of the categories defined in Table 4.3 for $\alpha \in \{0.25, 0.20, 0.15, 0.10, 0.05, 0.01\}$ for both SPECjvm98 and DaCapo combined using 30 samples in each of the prevalent methods.

When the significance level α decreases – and the confidence level increases – there is a shift in the conclusions. For example, if the prevalent approach uses a average value based on the same number of samples as the proposed rigorous approach, decreasing α shifts the classification from the lower two rows in Table 4.4 to the top row.

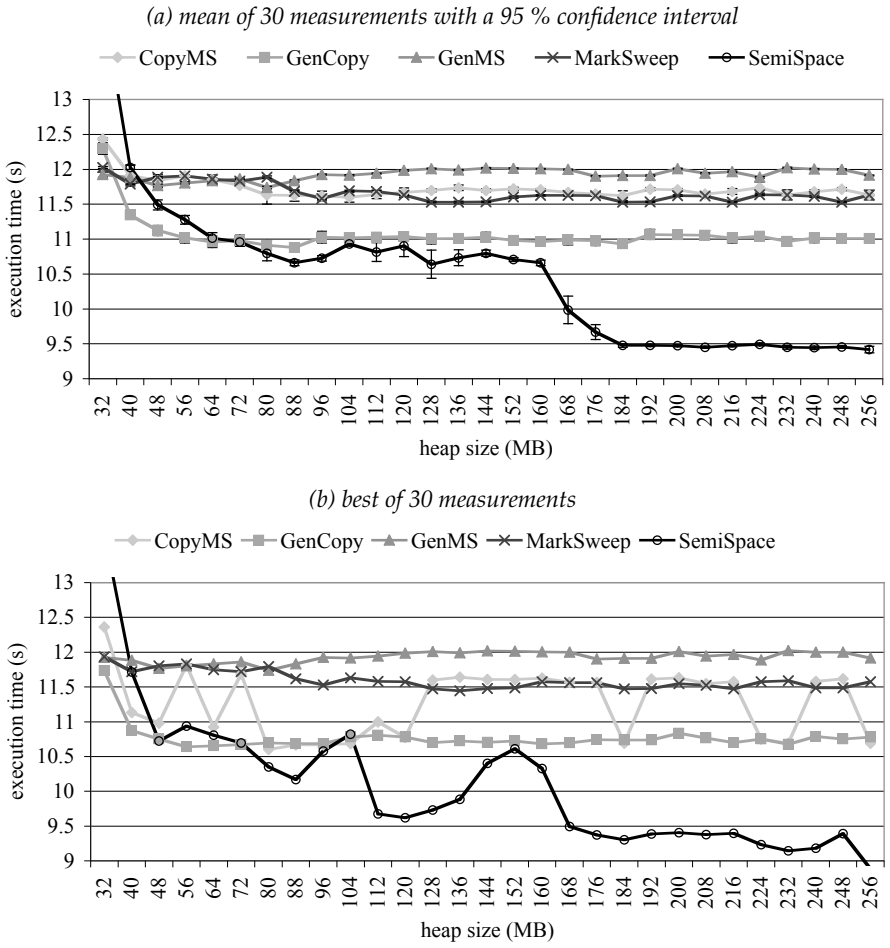


Figure 4.10: Start-up execution time (in seconds) for *db* as a function of heap size for five garbage collectors; mean of 30 measurements with 95 % confidence intervals (top) and best of 30 measurements (bottom).

4.5.2 Steady-state performance

We now consider steady-state performance. Figure 4.12 shows normalised execution time (averaged over a number of benchmarks) as a function of the number iterations for a single VM invocation. This graph shows that it takes a number of iterations before steady-state performance is reached: the first 3 iterations obviously seem to be part of start-up performance, and it takes more than 10 iterations before we actually reach steady-state performance.

For quantifying steady-state performance, following Section 4.3.2, we retain $k = 10$ iterations per VM invocation for which the CoV is smaller than

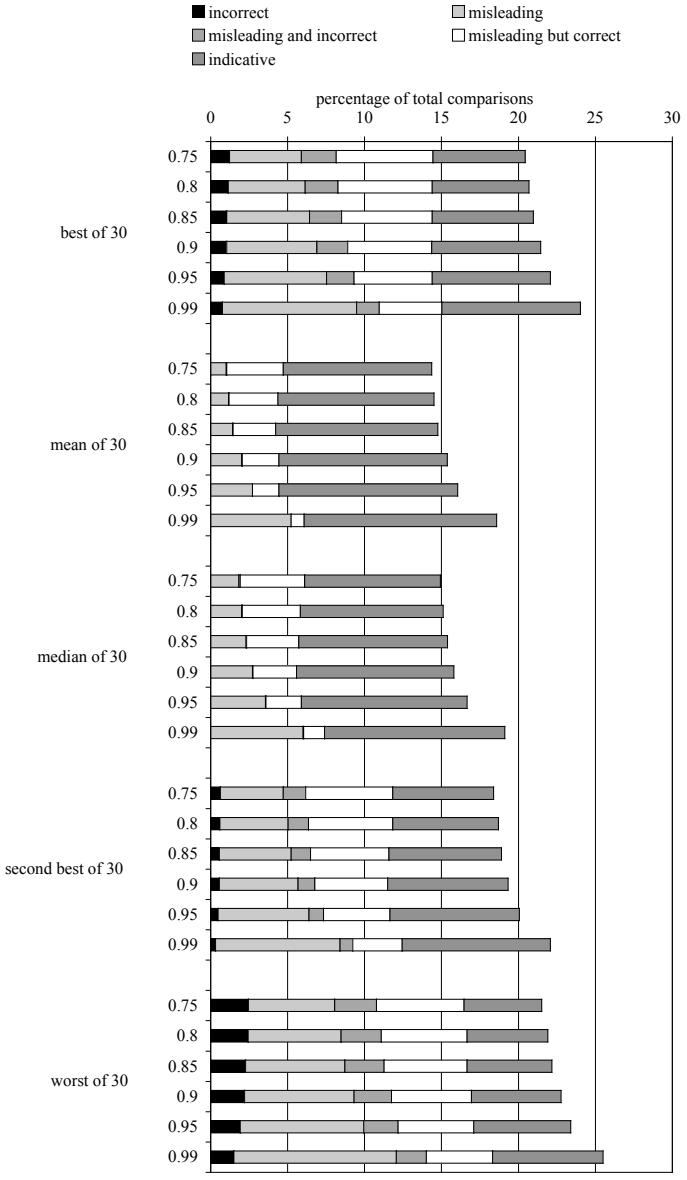


Figure 4.11: The effect of varying the significance level α on the decision classification for start-up execution. As the significance level decreases and the confidence level $1-\alpha$ increases, the number of comparisons that are not classified as correct increases. These results are obtained for SPECjvm98 and DaCapo using 30 VM invocations.

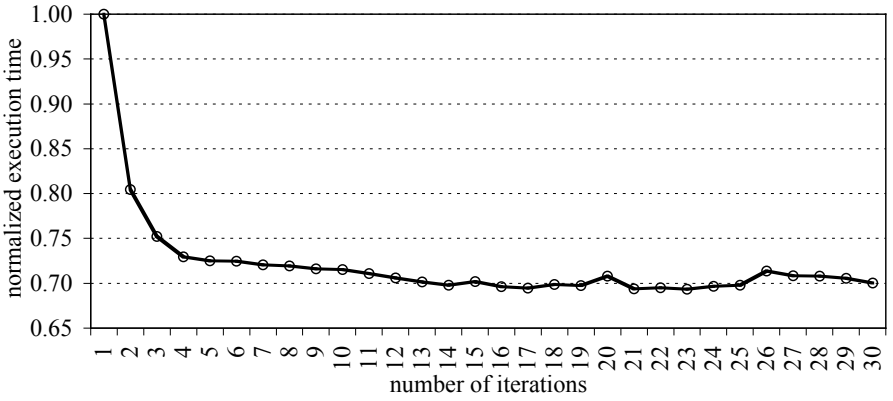


Figure 4.12: Normalised execution time as a function of the number of iterations on the AMD Athlon machine.

0.02. Figures 4.13 and 4.14 compare three prevalent steady-state performance methodologies against the statistically rigorous approach: (i) best of median (take the median per iteration across all VM invocations, and then select the best median iteration), (ii) best performance number, and (iii) second best performance number across all iterations and across all VM invocations. For these prevalent methods we consider 1, 3 and 5 VM invocations and 3, 5, 10 and 30 iterations per VM invocation. The general conclusion concerning the accuracy of the prevalent methods is similar to those for start-up performance. Prevalent methods are misleading in more than 20 % of the cases for a $\theta = 1\%$ threshold, more than 10 % for a $\theta = 2\%$ threshold, and more than 5 % for a $\theta = 3\%$ threshold. Also, the number of incorrect conclusions is not negligible (a few percent for small θ thresholds).

4.5.3 Replay compilation

As discussed before, replay compilation is a frequently used experimental design setup for comparing garbage collection alternatives. The reason for this is that it controls the non-determinism that is due to the the virtual machine's (time-based) adaptive compilation and optimisation system.

The goal of this section is twofold. First, we focus on experimental design and quantify how replay compilation compares against non-controlled compilation, assuming statistically rigorous data analysis. Second, we compare prevalent data analysis techniques against statistically rigorous data analysis under replay compilation.

In our replay compilation approach, we analyse 7 benchmark runs in separate VM invocations and take the optimal (yielding the shortest execution time) compilation plan. We also evaluated the majority plan and obtained

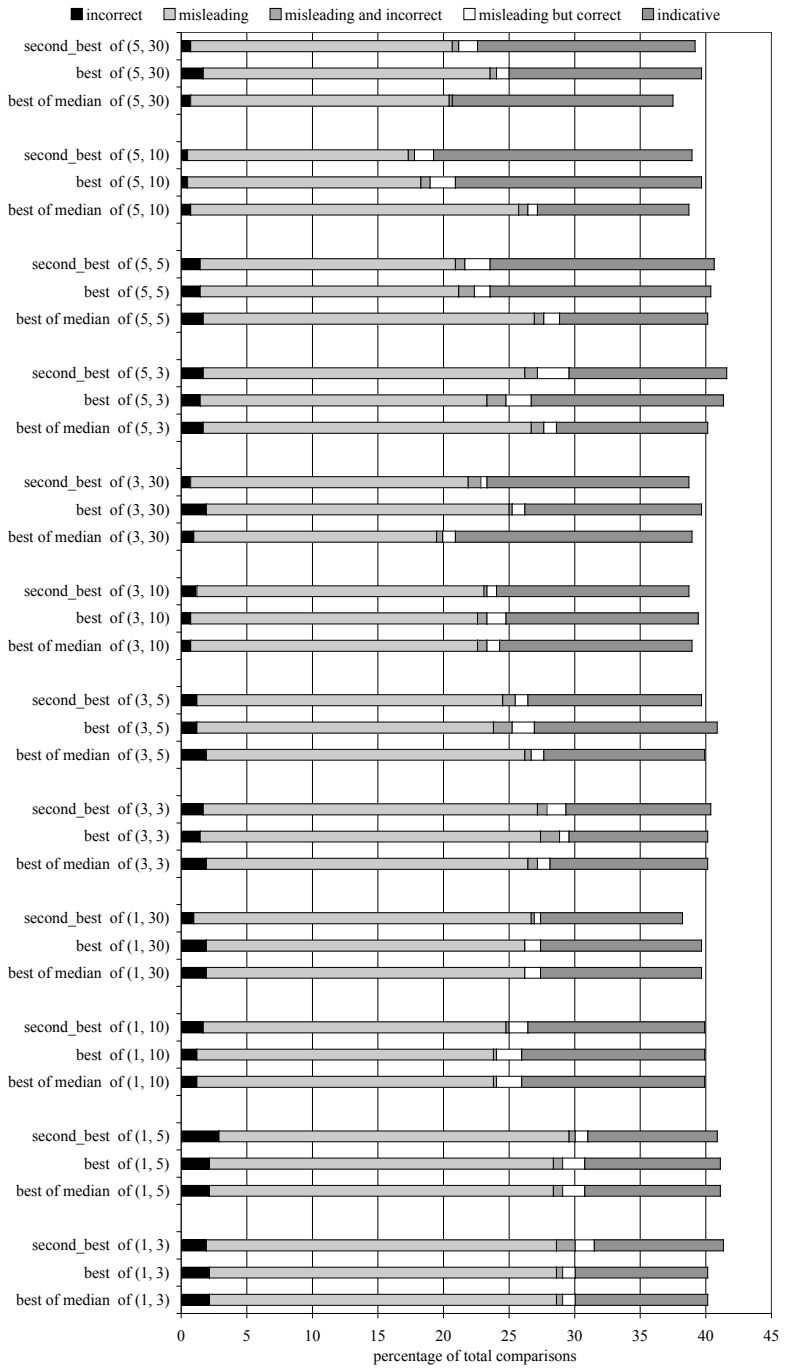


Figure 4.13: The (in)accuracy of prevalent methodologies compared to rigorous data analysis for steady-state performance: (x, y) denotes x VM invocations and y iterations per VM invocation; for SPECjvm98 on the AMD Athlon machine. The threshold $\theta = 1\%$.

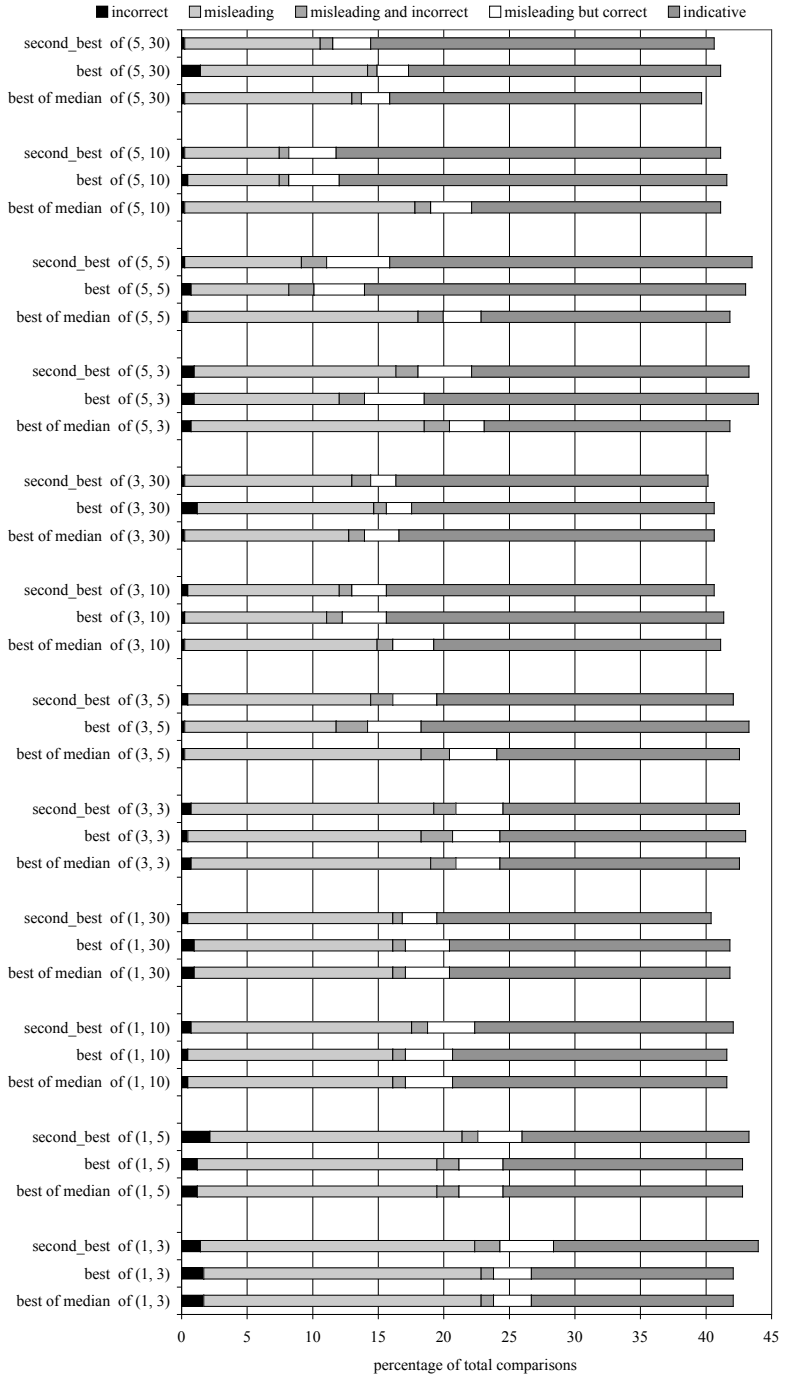


Figure 4.14: The (in)accuracy of prevalent methodologies compared to rigorous data analysis for steady-state performance: (x, y) denotes x VM invocations and y iterations per VM invocation; for SPECjvm98 on the AMD Athlon machine. The threshold $\theta = 2\%$.

Non-controlled compilation	**Replay compilation**		
	H_0^{r} is not rejected	H_0^{r} is rejected	
		$A > B$	$B > A$
H_0^{nc} is not rejected	*agree*	*inconclusive*	*inconclusive*
H_0^{nc} is rejected, $A > B$	*inconclusive*	*agree*	*disagree*
H_0^{nc} is rejected, $B > A$	*inconclusive*	*disagree*	*agree*

Table 4.4: Classifying conclusions by replay compilation versus non-controlled compilation. H_0^{nc} denotes the null hypothesis that the alternatives have equal means under the non-controlled compilation execution, H_0^{r} denotes the null hypothesis that the alternatives have equal means under the replay-compilation execution.

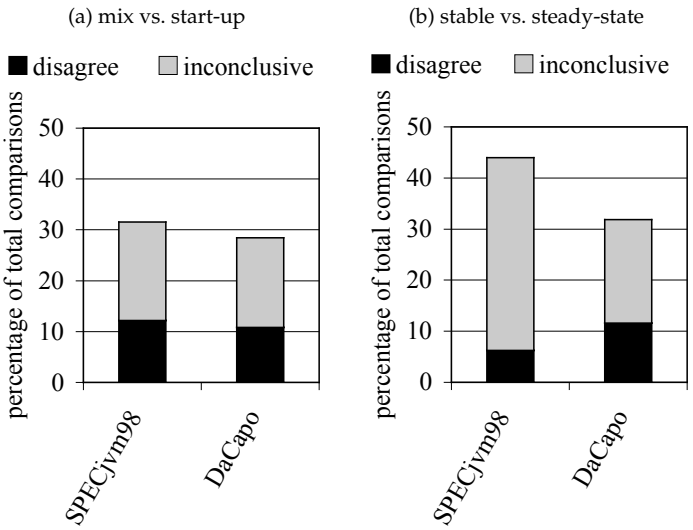


Figure 4.15: Comparing (a) mix replay compilation versus start-up performance, and (b) stable replay compilation versus steady-state performance under non-controlled compilation using statistically rigorous data analysis on the AMD Athlon XP platform.

similar results. The compilation plan is derived for start-up performance – determining the plan after a single iteration – using the GenMS configuration with a heap size that is 8 times the minimal heap size for each benchmark. The timing run consists of two benchmark iterations: the first one (mix), includes compilation activity, and the second one (stable) does not include compilation activity. A full GC is performed between these two iterations. The timing runs are repeated multiple times (3, 5, 10 and 30 times in our setup).

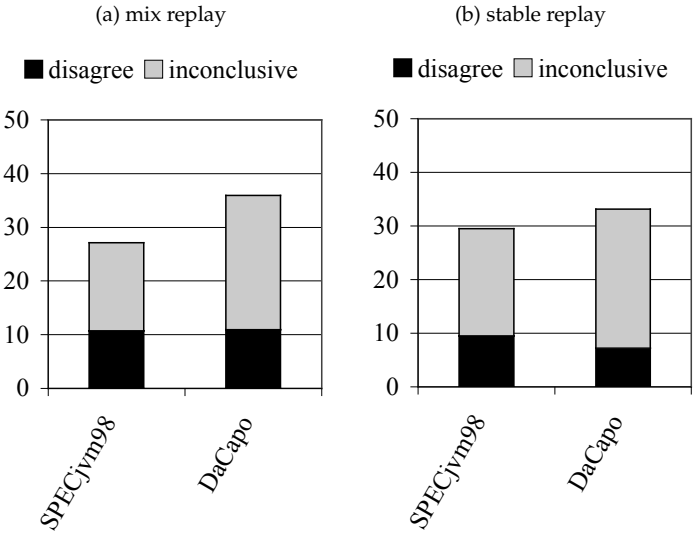


Figure 4.16: Percentage of disagreeing and inconclusive comparisons under (a) mix replay and (b) stable replay for SPECjvm98 and DaCapo when comparing majority plans versus a non-controlled execution using ANOVA plus a Tukey HSD post-hoc test with a 95 % confidence level or a 5 % significance level on the AMD Athlon XP.

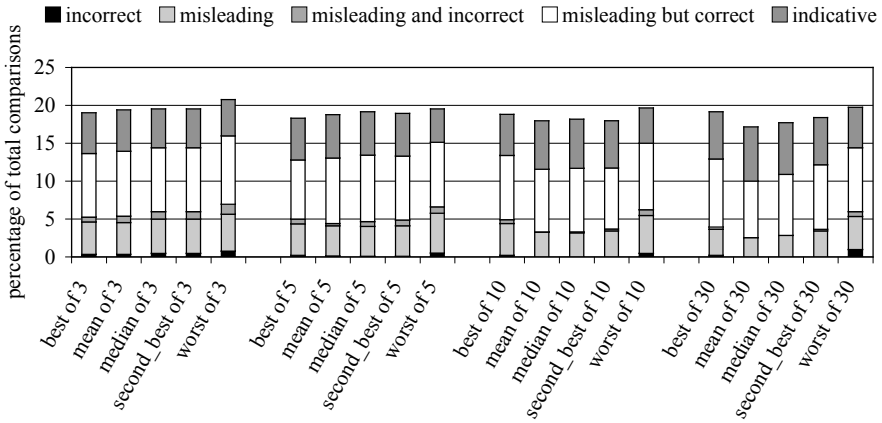


Figure 4.17: Comparing prevalent data analysis versus statistically rigorous data analysis under mix replay compilation, assuming $\theta = 1\%$ on the AMD Athlon XP platform.

Experimental design. Figure 4.15 compares mix replay versus start-up performance as well as stable replay versus steady-state performance, assuming non-controlled compilation and using an optimal compilation plan for replay.

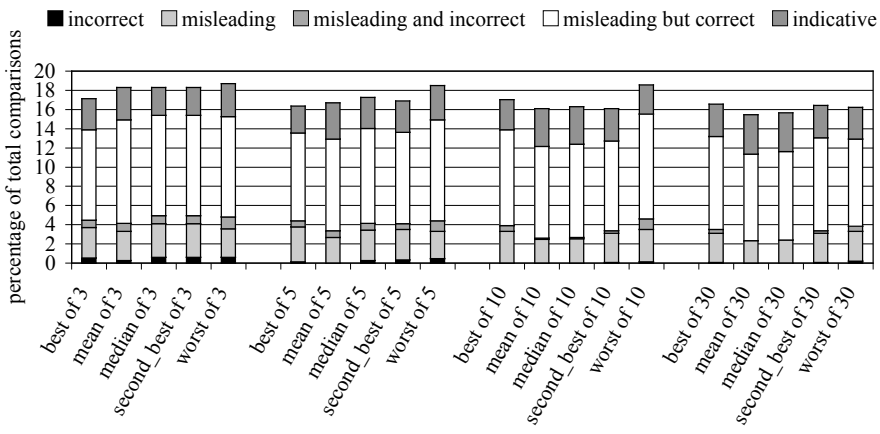


Figure 4.18: Comparing prevalent data analysis versus statistically rigorous data analysis under stable replay compilation, assuming $\theta = 1\%$ on the AMD Athlon XP platform.

We assume statistically rigorous data analysis for both the replay compilation and non-controlled compilation experimental setups. We classify all GC comparisons in three categories: *agree*, *disagree* and *inconclusive*, see Table 4.4, and display the *disagree* and *inconclusive* categories in Figure 4.15. We observe replay compilation and non-controlled compilation agree in 56 % to 72 % of all cases, and are inconclusive in 17 % (DaCapo mix versus startup) to 37 % (SPECjvm98 stable versus steady-state) of all cases. In up to 12 % of all cases, see SPECjvm98 mix versus start-up and DaCapo stable versus steady-state, both experimental designs disagree. These two experimental designs offer different garbage collection loads and thus expose different space-time tradeoffs that the collectors make. Additionally, Figure 4.16 illustrates that majority plans do not always match the results obtained from performing a non-controlled experiment, i.e., without replay compilation. The graph shows the number of disagreeing and inconclusive comparisons between both types of experiments. Under mix replay, up to 10.7 % of all conclusions are contradictory, and up to 25 % are inconclusive. For stable replay these numbers become 9.5 % and 25.9 %, respectively.

Data analysis. We now assume replay compilation as the experimental design setup, and compare prevalent data analysis versus statistically rigorous data analysis. Figures 4.17 and 4.18 show the results for mix replay versus start-up performance, and stable replay versus steady-state performance, respectively. These results show that prevalent data analysis can be misleading under replay compilation for start-up performance: the fraction misleading conclusions is around 5 %, see Figure 4.17. For steady-state performance, the number of misleading conclusions is less than 4 %, see Figure 4.18.

We will address replay compilation in more depth in the next chapter, as there are some interesting opportunities to improve the statistical rigour in the data analysis for this particular experimental design.

4.6 JavaStats: statistically rigorous performance evaluation in practise

As discussed in Section 4.2, the width of the confidence interval is a function of the number of measurements n. In general, the width of the confidence interval decreases with an increasing number of measurements as shown in Figure 4.19. The width of the 95 % confidence interval is shown as a percentage of the mean sample value (on the vertical axis) as a function of the number of measurements taken (on the horizontal axis). We show three example benchmarks: *jess*, *db* and *mtrt* for a 80 MB heap size on the AMD Athlon machine. The various curves represent different garbage collectors for start-up performance. The interesting observation here is that the width of the confidence interval largely depends on both the benchmark and the garbage collector. For example, the width of the confidence interval for the GenCopy collector for *jess* is fairly large, more than 3 %, even for 30 measurements. For the MarkSweep and GenMS collectors for *db* on the other hand, the confidence interval is much smaller, around 1 % even after less than 10 measurements.

These observations motivated us to come up with an automated way of determining how many measurements are needed to achieve a desired confidence interval width. For example, for *db* and the MarkSweep and GenMS collectors, a handful of measurements will suffice to achieve a very small confidence interval, whereas for *jess* and the GenCopy collector many more measurements are needed. We provide publicly available software called JavaStats [5] that readily works with the SPECjvm98 and DaCapo benchmark suites, and which facilitates the application of our start-up and steady-state evaluation methodology. For start-up performance, a script (i) triggers multiple VM invocations running a single benchmark iteration, (ii) monitors the execution time of each invocation, and (iii) computes the confidence interval for a given confidence level.

Figure 4.20 reports the number of VM invocations required for start-up performance to achieve a 2 % confidence interval width with a maximum number of VM invocations, $p = 30$ for *jess*, *db* and *mtrt* on the AMD Athlon as a function of heap size for the five garbage collectors. The interesting observation here is that the number of measurements taken varies from benchmark to benchmark, from collector to collector and from heap size to heap size. This once again shows why an automated way of collecting measurements is desirable. Having to take fewer measurements for a desired level of confidence speeds up the experiments compared to taking a fixed number of measure-

[5] Available at `http://www.elis.UGent.be/JavaStats/`.

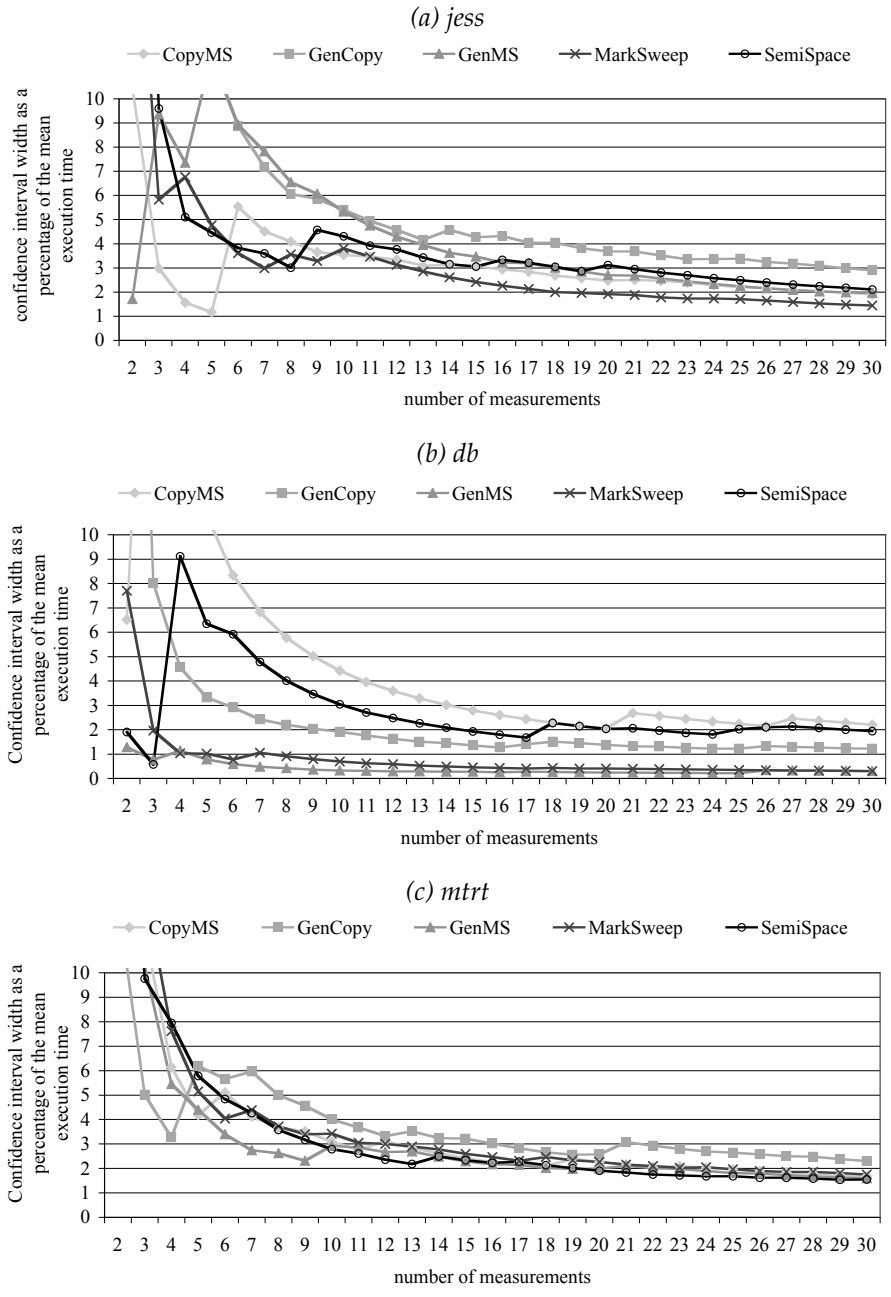


Figure 4.19: Confidence width as a percentage of the mean (on the vertical axis) as a function of the number of measurements taken (on the horizontal axis) for three benchmarks: *jess* (top), *db* (middle) and *mtrt* (bottom).

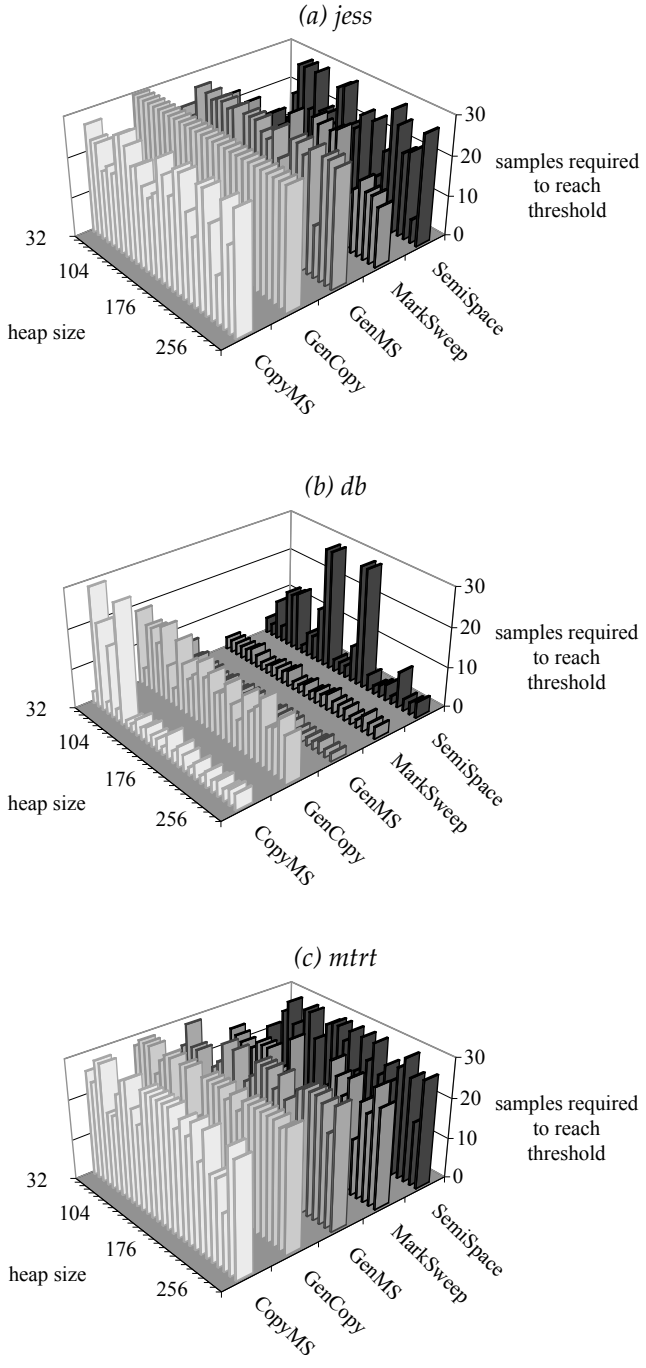


Figure 4.20: Figure shows how many measurements are required before reaching a 2 % confidence interval on the AMD Athlon machine.

ments.

For steady-state performance, JavaStats collects execution times across multiple VM invocations and across multiple benchmark iterations within a single VM invocation. JavaStats consists of a script running multiple VM invocations as well as a benchmark harness triggering multiple iterations within a single VM invocation. The output for steady-state performance is similar to what is reported above for start-up performance.

SPECjvm98 as well as the DaCapo benchmark suite already come with a harness to set the desired number of benchmark iterations within a single VM invocation. The current version of the DaCapo harness also determines how many iterations are needed to achieve a desired level of coefficient of variation (CoV). As soon as the observed CoV drops below a given threshold (the convergence target) for a given window of iterations, the execution time for the next iteration is reported. JavaStats extends the existing harnesses (i) by enabling measurements across multiple VM invocations instead of a single VM invocation, and (ii) by computing and reporting confidence intervals.

A final note that we would like to make is that collecting the measurements for a statistically rigorous data analysis can be time-consuming, especially if the experiment needs a large number of VM invocations and multiple benchmark iterations per VM invocation (in case of steady-state performance). Under time pressure, statistically rigorous data analysis can still be applied considering a limited number of measurements, however, the confidence intervals will be wider.

4.7 Conclusion

In this chapter we have shown there is a pitfall associated with prevalent performance data analysis methodologies. For one particular setup in which we compare performance of five garbage collection algorithms, we found that up to 20 % conclusions made by prevalent approaches are either misleading or incorrect. This pitfall can be avoided by using a rigorous analysis. The analysis presented in this chapter does not necessarily invalidate the majority of conclusions made in the past, but illustrates that some of the conclusions may be misleading or incorrect.

Chapter 5

Replay compilation revisited

This chapter revisits replay compilation and proposes improvements for its experimental design and data analysis.

5.1 Introduction

Current practice in replay compilation typically uses a single compilation plan during replay.

In this chapter, we argue that the performance results obtained from a single compilation plan may not be representative for other compilation plans, and may potentially yield misleading results. We therefore advocate considering multiple compilation plans in order to better represent average behaviour. Given the use of multiple plans, we can adapt the statistical data analysis to benefit from it. Indeed, using multiple plans allows employing a matched-pair comparison as a statistically rigorous data analysis technique for comparing design alternatives under replay compilation. Such an analysis amortises part of the overhead introduced by multiple compilation plans.

We make the following contributions:

- We show that different compilation plans lead to statistically significant execution time variability. The reason is that different compilation plans may lead to different methods being compiled to different optimisation levels. And this execution time variability may lead to inconsistent conclusions across compilation plans in practical research studies.
- We advocate replay compilation using multiple compilation plans in order to capture the execution time variability across compilation plans. Multiple compilation plans result in a more rigorous replay compilation methodology compared to prior work which considers a single compilation plan during replay.

- We propose matched-pair comparison for analysing the performance numbers obtained from replay compilation using multiple compilation plans. Matched-pair comparison considers the performance numbers for a given compilation plan before and after the innovation as a pair. In general, this yields tighter confidence intervals than statistical analysis assuming unpaired measurements. Or, for the same level of accuracy, i.e., for the same confidence interval size, fewer compilation plans are to be considered under matched-pair comparison.
- We leverage this important property and demonstrate that for a given experimentation time budget, it is beneficial to consider more compilation plans rather than more runs per compilation plan.

This chapter is organised as follows. Section 5.2 very briefly touches the experimental setup we use for the experiments described in this chapter. In Section 5.3, we evaluate the run time variability across compilation plans at replay time, and how this may affect conclusions in research studies. Section 5.4 then presents matched-pair comparison for the statistical analysis of performance numbers obtained from multiple compilation plans. In Section 5.5, we then describe the overall framework for rigorous replay compilation using multiple compilation plans. We finally conclude in Section 5.6.

5.2 Replay compilation setup

As Jikes RVM employs timer-based sampling to detect optimisation candidates, researchers have implemented replay compilation in Jikes RVM to control non-determinism using so-called advice files. An advice file specifies (i) the optimisation level for each method compiled, (ii) the dynamic call graph profile, and (iii) the edge profile. Advice files are collected through a profile run: through command-line arguments, Jikes RVM can be instructed to generate an advice file while the program executes. Then, in the replay run, Jikes RVM compiles each method in the advice file to the specified level upon a method's first invocation. If there is no advice for a method, the method is compiled using Jikes RVM's baseline compiler.

In our setup, the compilation plans are computed by running a benchmark on the Jikes RVM using the GenMS garbage collector and a heap size that is 8 times the minimum heap size. We compute a different compilation plan within a single virtual machine invocation after (i) a single benchmark iteration (further referred to as a 1-iteration plan), and (iii) ten benchmark iterations (further referred to as a 10-iteration plan). These are experimental design choices; the literature either indicates the usage of choice (i) or does not make any indication at all. Additionally, compilation plans are computed separately for each hardware platform.

For each particular configuration, i.e., a benchmark, a compilation plan, a heap size, and a garbage collector, we measure 11 executions, and drop the

first measurement. In these experiments, we use the same setup as in Chapter 4.

5.3 A single compilation plan or multiple compilation plans?

Having detailed our experimental setup, we now evaluate replay compilation. In particular, we study the accuracy of selecting a single compilation plan for driving replay compilation. This study will make the case for multiple compilation plans instead of a single compilation plan.

This is done in four steps. We first demonstrate that different compilation plans can lead to statistically significantly different benchmark execution times. This is the case for both GC time and mutator time. Second, we provide a reason for this difference in execution time, by comparing the methods that are compiled under different compilation plans. Third, we present a case study in which we compare various garbage collection strategies using replay compilation as the experimental design setup. This case study demonstrates that the conclusions taken from practical research studies may be subject to the chosen compilation plan. Finally, we demonstrate that a majority plan (which combines multiple profiles) is no substitute for multiple compilation plans.

5.3.1 Execution time variability

We first study how benchmark execution time is affected by the compilation plan under replay compilation. To do so, we consider the following experiment. First, we collect 10 compilation plans per benchmark – this is done by profiling the benchmark execution 10 times. Second, we consider 5 GC strategies and 6 heap sizes per benchmark, as explained in the previous section; and for each GC strategy and heap size combination, we run each benchmark 10 times for each of the compilation plans. This yields 100 execution times in total per combination of benchmark, GC strategy and heap size. The goal of this experiment is to quantify whether the execution time variability observed across these 100 measurements is determined more by the compilation plan than by the runtime variability per compilation plan.

Example 5.1. *Figure 5.1 illustrates this experiment for a typical benchmark, namely jython – we observed similar results for other benchmarks. Violin plots are displayed which show the GC time and mutator time variability (on the vertical axis) both within and across compilation plans (on the horizontal axis). The middle point in a violin plot shows the median, and the shape of the violin plot represents the distribution's probability density function: the wider the violin plot, the higher the density. The top and bottom points show the maximum and minimum values. This figure suggests that the variability within a compilation plan is much smaller than the variability across compilation plans.*

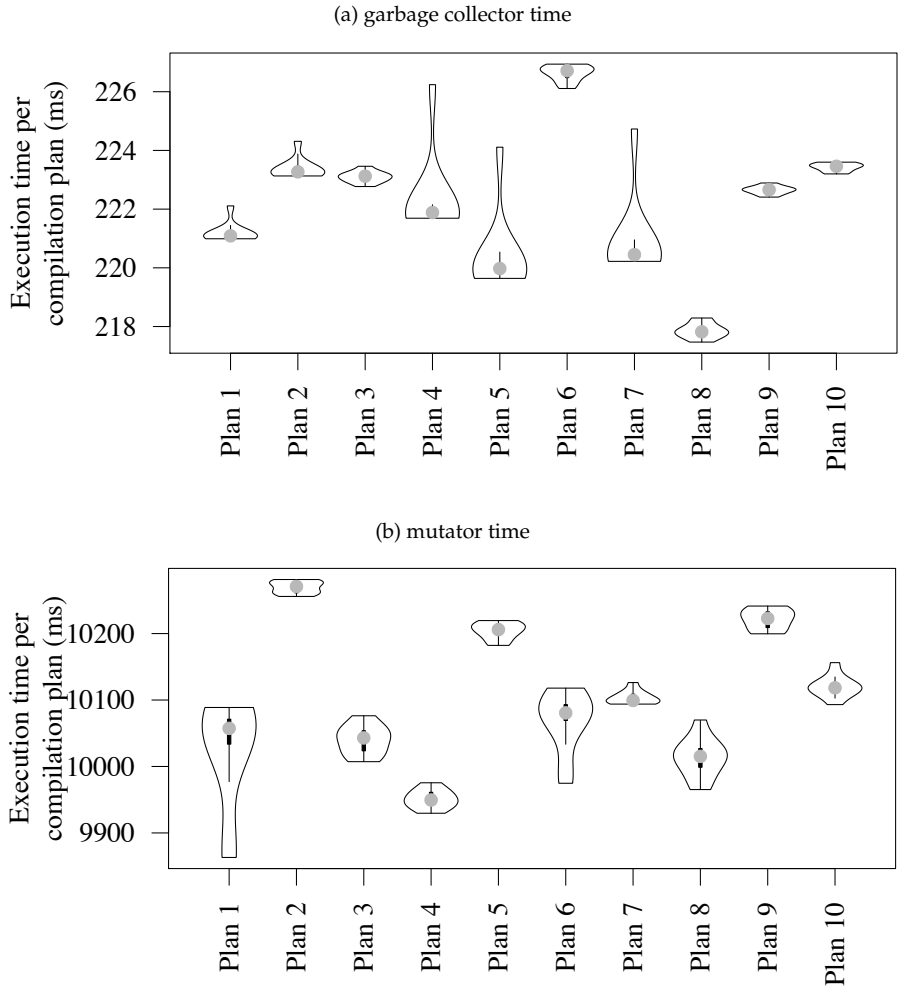


Figure 5.1: Violin plots illustrating the variability in (a) GC time and (b) mutator time within and across compilation plans for *jython* on the AMD Athlon, the GenMS garbage collector, and a 144 MB heap size; assuming a stable run and a 10-iteration compilation plan. The time is given in milliseconds, the difference between the highest and lowest value for GC time is 3.6% and for mutator time it is 6.2%.

To study the execution time variability across compilation plans in a more statistically rigorous manner, we use a single-factor ANOVA [62, 71, 82] in which the compilation plans are the alternatives. To recapitulate, ANOVA separates the total variation observed in (i) the variation observed *within* each alternative, which is assumed to be a result of random effects in the measurements, and (ii) the variation *between* the alternatives. If the variation between the alternatives is larger than the variation within each alternative, then it can

be concluded that there is a statistically significant difference between the alternatives. In this experiment, the alternatives are the compilation plans, and thus, ANOVA will figure out whether the execution time variability in our experiment is due to random effects rather than due to the compilation plans. ANOVA assumes that the variance in measurement error is the same for all of the alternatives. Also, ANOVA assumes that the errors in the measurements for the different alternatives are independent and normally distributed. However, ANOVA is fairly robust with respect to non-normally distributed measurements, especially in case there is a balanced number of measurements for each of the alternatives. Given the latter and the fact that we have 10 measurements per garbage collection strategy, we can use ANOVA for our purpose.

Figure 5.2 shows the percentage of the 30 experiments per benchmark (there are 5 GC strategies and 6 heap sizes) for which the ANOVA reports there is a statistically significant difference in total execution time at the 95% confidence level between the various compilation plans for a 1-iteration plan. The top graph in Figure 5.2 shows the mix run results, whereas the bottom graph shows the stable run results; there are two bars per benchmarks for the Intel Pentium 4 and AMD Athlon machines, respectively. For the majority of the benchmarks, there is a statistically significant difference in execution times across multiple compilation plans. For several benchmarks, the score equals 100% which means that all the compilation plans are significantly different from each other. The difference tends to be higher for the mix runs than for the stable runs for most of the DaCapo benchmarks on the AMD platform. This suggests that once the methods are compiled and optimized in the mix run, performance seems to be more similar across compilation plans in the stable run.

Figure 5.3 shows similar results for the 10-iteration compilation plans, but now we make a distinction between GC, mutator and total time, and we assume the stable run. We conclude that even under 10-iteration compilation plans there still is a large fraction of experiments for which we observe statistically significant differences across the compilation plans. And this is the case for GC, mutator and total time.

5.3.2 Compilation load variability

Now that we have shown that different compilation plans can result in statistically significantly different execution times, this section aims at quantifying why this is the case. Our intuition tells us that the varying execution times are due to different methods being compiled at different levels of optimisation across compilation plans. To support this hypothesis we quantify the relative difference in compilation plans.

To do so, we determine the Method Optimisation Vector (MOV) per compilation plan. Each entry in the MOV represents an optimised method along with its (highest) optimisation level; the MOV does not include an entry for

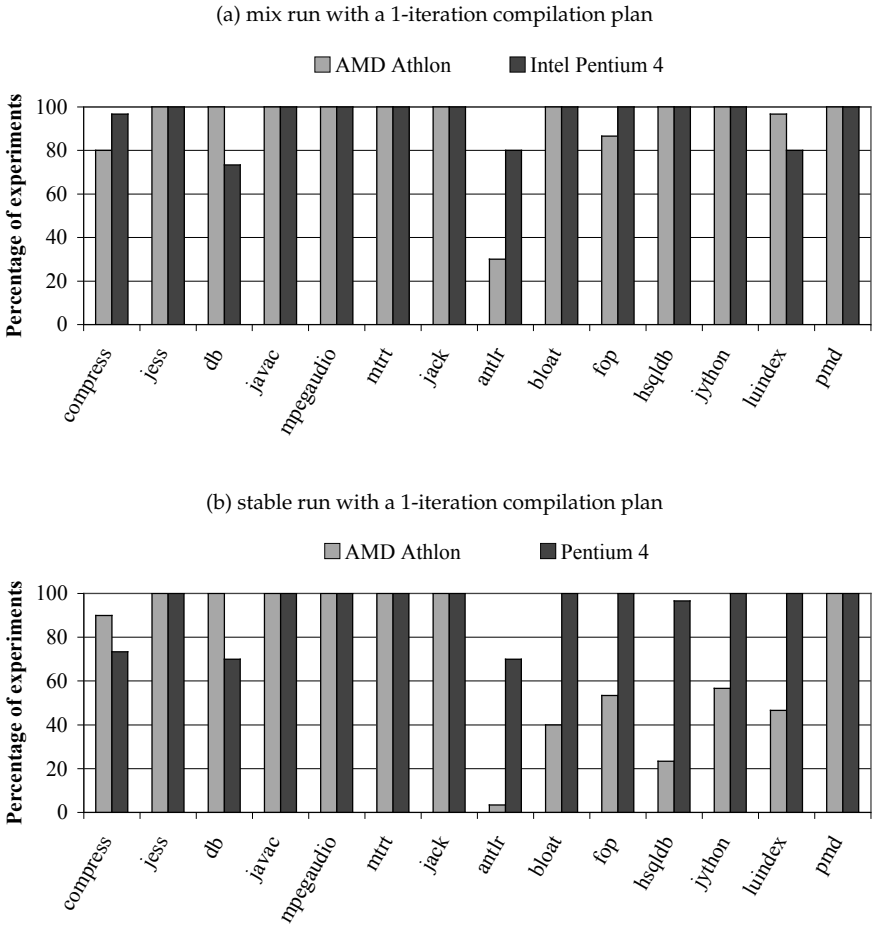


Figure 5.2: The percentage of experiments per benchmark, for which there is a statistically significant difference in total execution time between runs for each of the ten 1-iteration compilation plans on the AMD Athlon XP and the Intel Pentium 4 platforms. The top and bottom graphs show results for mix and stable replay, respectively.

baseline compiled methods. For example, if in one compilation plan, method `foo_1` gets optimised to level 1, method `foo_2` gets optimised to level 0, and method `foo_3` gets only baseline compiled, then the MOV looks like `[(foo_1,1);(foo_2,0)]`. In another compilation plan, method `foo_1` gets optimised to optimisation level 1 as well, whereas method `foo_2` gets baseline compiled, and `foo_3` gets optimised to level 0, then the MOV looks like `[(foo_1,1);(foo_3,0)]`. Comparing the two compilation plans can then be done by comparing their respective MOVs. This is done by counting the number of (method, optimisation level) pairs that appear in both MOVs, divided by the total number of methods appearing in both compilation plans,

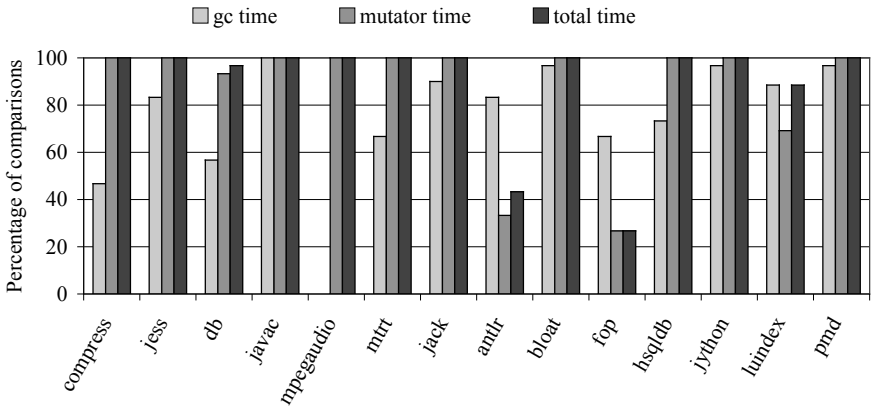


Figure 5.3: The fraction of experiments per benchmark, for which there is a statistically significant difference in GC, mutator and total time across compilation plans. These graphs assume 10-iteration plans and stable runs on the AMD Athlon XP.

i.e., an unweighted overlap [30]. The overlap metric varies between 0 and 1, with 0 meaning there is no overlap and 1 meaning there is perfect overlap. In the above example, the overlap metric equals 1/3, i.e., there is one common (method, optimisation level) pair that appears in both MOVs, namely `(foo,1)` and there are three methods optimised in at least one of the compilation plans.

Figure 5.4 quantifies the overlap metric per benchmark computed as an average across all (unique) pairs of 10 compilation plans – there are $C_{10}^2 = 45$ unique pairs of compilation plans over which the average overlap metric is computed. We observe that the overlap is rather limited, typically under 0.4 for most of the benchmarks. There are a couple benchmarks with relatively higher overlap metrics, see for example *compress* and *db*. These benchmarks have a small code footprint and therefore there is a higher probability that the same methods will get sampled across multiple profiling runs of the same benchmark. We conclude that the significant performance differences across compilation plans are due to compilation load differences.

5.3.3 Case study: Comparing GC strategies

We now study whether different compilation plans can lead to different conclusions in practical research studies. In order to do so, we consider a case study that compares GC strategies using replay compilation as the experimental design – this reflects a widely used methodology in GC research, see for example [15, 18, 19, 53, 59, 92, 93, 101, 110]. GC poses a complex space-time

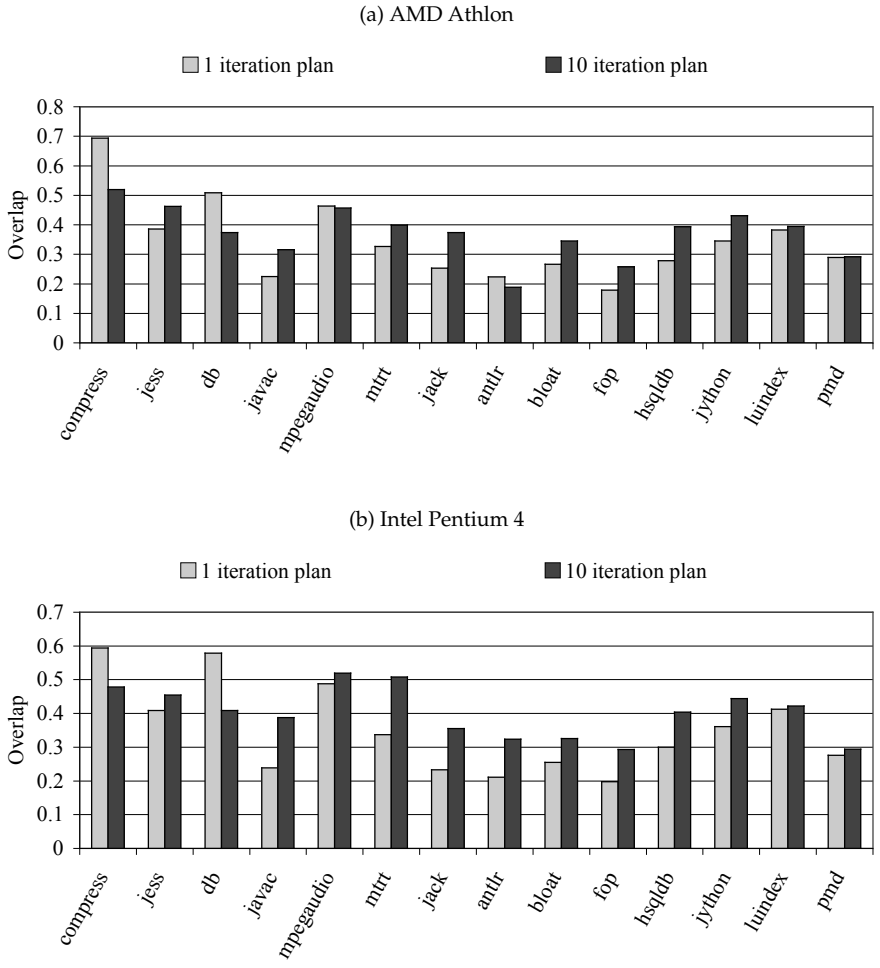


Figure 5.4: Average overlap across compilation plans on (a) the AMD Athlon platform, and (b) the Intel Pentium 4 platform, for the 1-iteration and 10-iteration compilation plans.

trade-off, and it is unclear which GC strategy is the winner without detailed experimentation.

We use the same data set as before. There are 14 benchmarks (7 SPECjvm98 benchmarks and 7 DaCapo benchmarks), and we consider 5 GC strategies and 6 heap sizes per benchmark. For each benchmark, GC strategy and heap size combination, we have 10 measurements per compilation plan for both the mix and stable runs; and we consider 1-iteration and 10-iteration plans. We then compute the average execution time along with its 95% confidence interval across these 10 measurements, following the statistically rigorous methodol-

compilation plan j	compilation plan i		
	H_0^i is not rejected	H_0^i is rejected	
		$A > B$	$B > A$
H_0^j is not rejected	*agree*	*inconclusive*	*inconclusive*
H_0^j is rejected, $A > B$	*inconclusive*	*agree*	*disagree*
H_0^j is rejected, $B > A$	*inconclusive*	*disagree*	*agree*

Table 5.1: Classifying GC comparisons when comparing compilation plans.

ogy detailed in Chapter 4 – specifically, we use ANOVA in conjunction with the Tukey HSD test to compute the simultaneous 95% confidence intervals. This yields the average execution time along with its confidence interval per GC strategy and heap size, for each benchmark and compilation plan. We then compare these averages and confidence intervals by doing a pairwise comparison across compilation plans. The goal of this comparison is to verify whether different compilation plans lead to consistent conclusions about the best GC strategy for a given heap size.

When comparing two compilation plans, we compare the execution times per pair of GC strategies (per heap size) and classify this comparison in one of the three categories: *agree*, *disagree* and *inconclusive*, see also Table 4.4. For a given compilation plan p_i and GC strategies A and B, we define the null hypothesis as $H_0^i \equiv \mu_{p_i}^A = \mu_{p_i}^B$. The null hypothesis states that GC strategies A and B achieve the same mean execution time under compilation plan p_i. Hence, if the null hypotheses H_0^i and H_0^j for compilation plans p_i and p_j, respectively, are rejected, and if in both cases the same GC strategy outperforms the other, then the comparison is classified as *agree*. This means that both compilation plans agree on the fact that GC strategy A outperforms GC strategy B (or vice versa) in a statistically significant way. In case both compilation plans yield the result that both GC strategies are statistically indifferent, i.e., for neither compilation plan the null hypothesis is rejected, we also classify the GC comparison as an *agree*. If on the other hand both compilation plans disagree on which GC strategy outperforms the other one, then we classify the comparison as *disagree*. In case the null hypothesis is rejected for one compilation plan, but not for the other, we classify the GC comparison as *inconclusive*.

1-iteration plans.

Figure 5.5 shows this classification per benchmark for the total execution time under mix and stable replay for 1-iteration compilation plans. The *disagree* and *inconclusive* categories are shown as a percentage – the *agree* category then is the complement to 100%. For several benchmarks, the fraction disagree comparisons is higher than 5%, and in some cases even higher than 10%. The *mpegaudio* benchmark is a special case with a very high disagree fraction al-

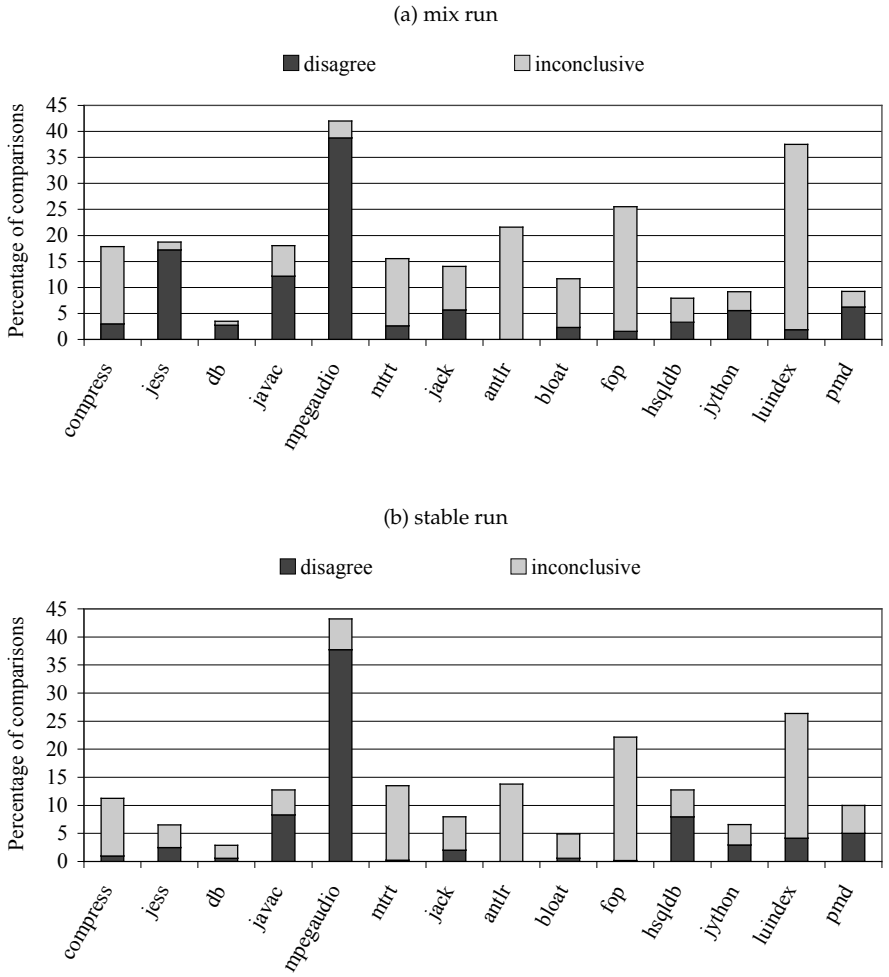


Figure 5.5: Percentage inconclusive and disagreeing comparisons on the AMD Athlon using 1-iteration compilations plans, under (a) mix replay and (b) stable replay.

though it has a very small live data footprint: the reason is that the various GC strategies affect performance through their heap data layout – see further for a more rigorous analysis. For many benchmarks, the fraction inconclusive comparisons is larger than 10%, for both the mix and stable runs, and up to 20% and higher for several benchmarks. In other words, in a significant number of cases, different compilation plans do not agree on which GC strategy performs best.

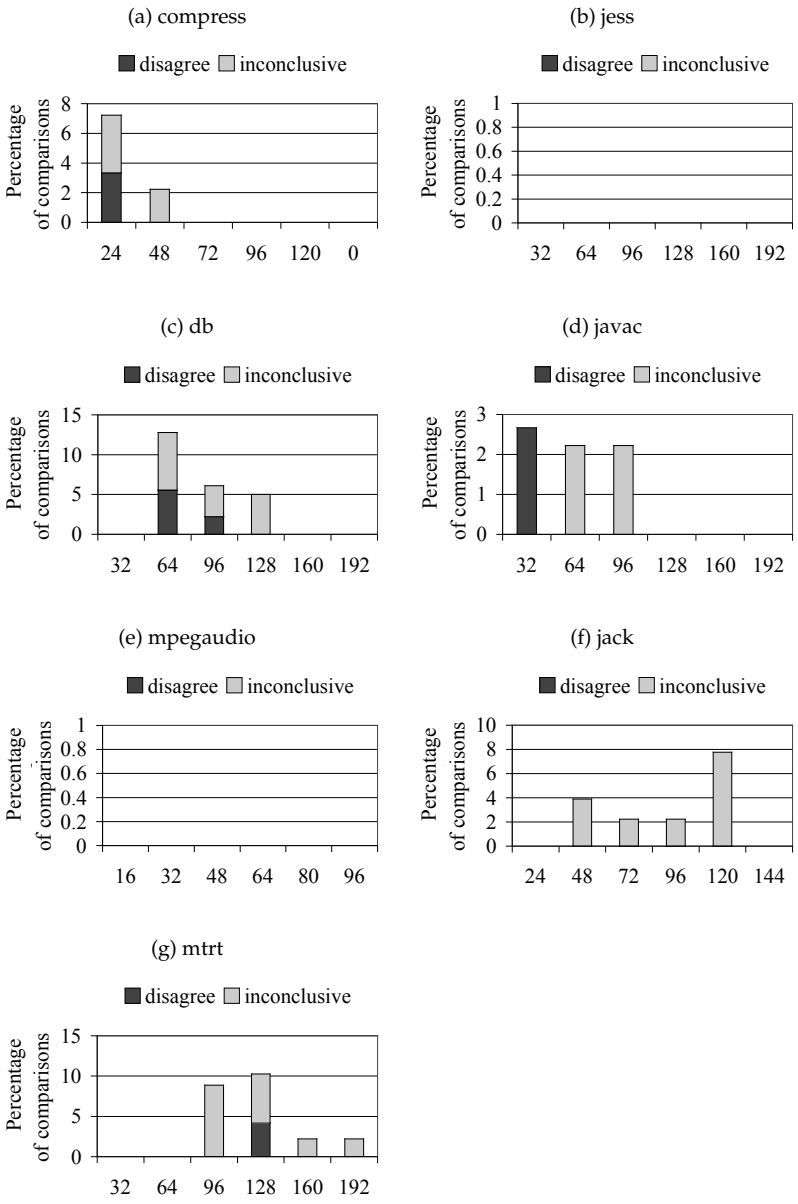


Figure 5.6: Percentage inconclusive and disagreeing comparisons for GC time under stable replay for SPECjvm98; heap size appears on the horizontal axis in each of the per-benchmark graphs.

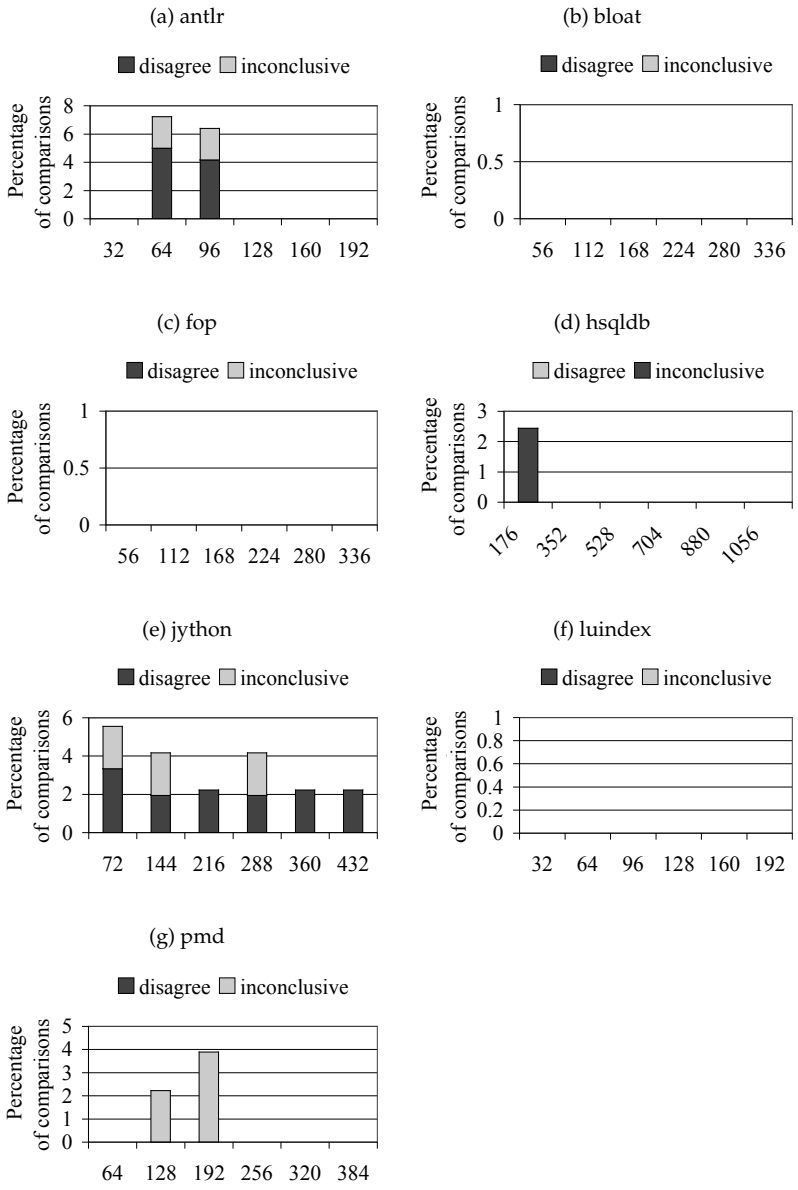


Figure 5.7: Percentage inconclusive and disagreeing comparisons for GC time under stable replay for DaCapo; heap size appears on the horizontal axis in each of the per-benchmark graphs.

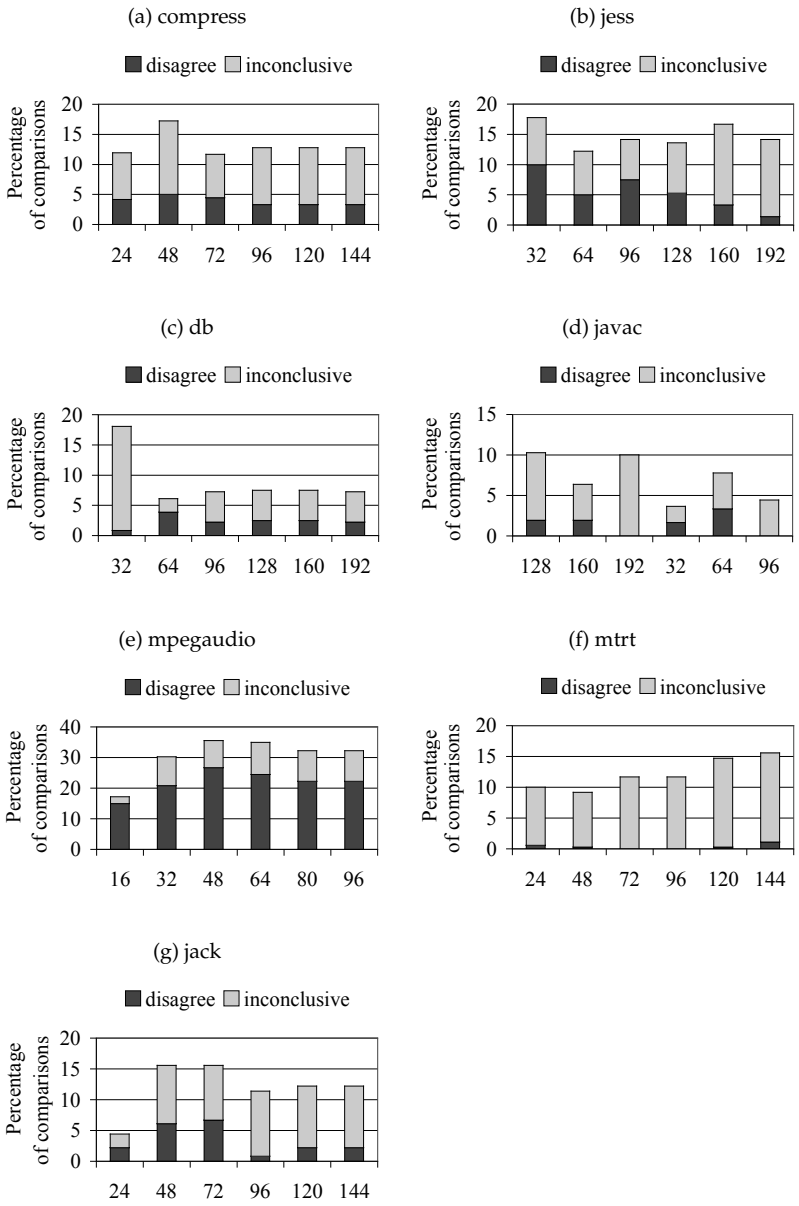


Figure 5.8: Percentage inconclusive and disagreeing comparisons for mutator time under stable replay for SPECjvm98; heap size appears on the horizontal axis in each of the per-benchmark graphs.

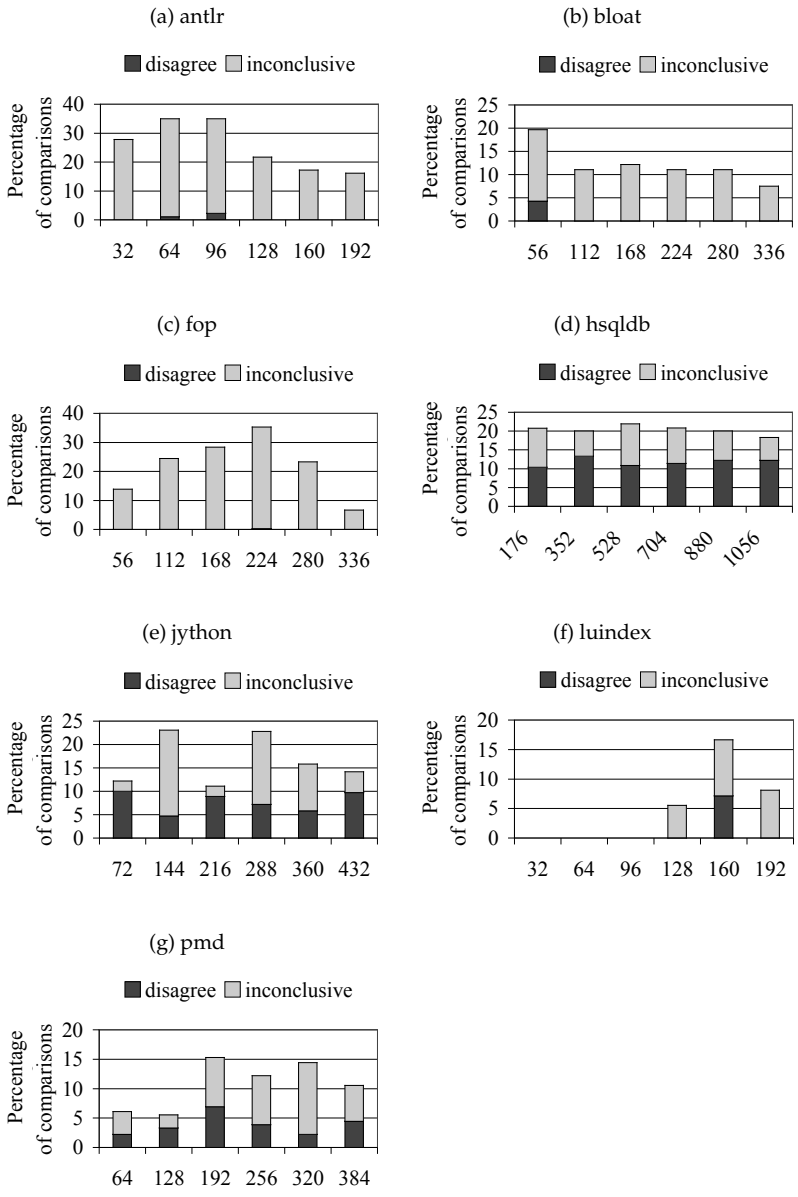


Figure 5.9: Percentage inconclusive and disagreeing comparisons for mutator time under stable replay for DaCapo; heap size appears on the horizontal axis in each of the per-benchmark graphs.

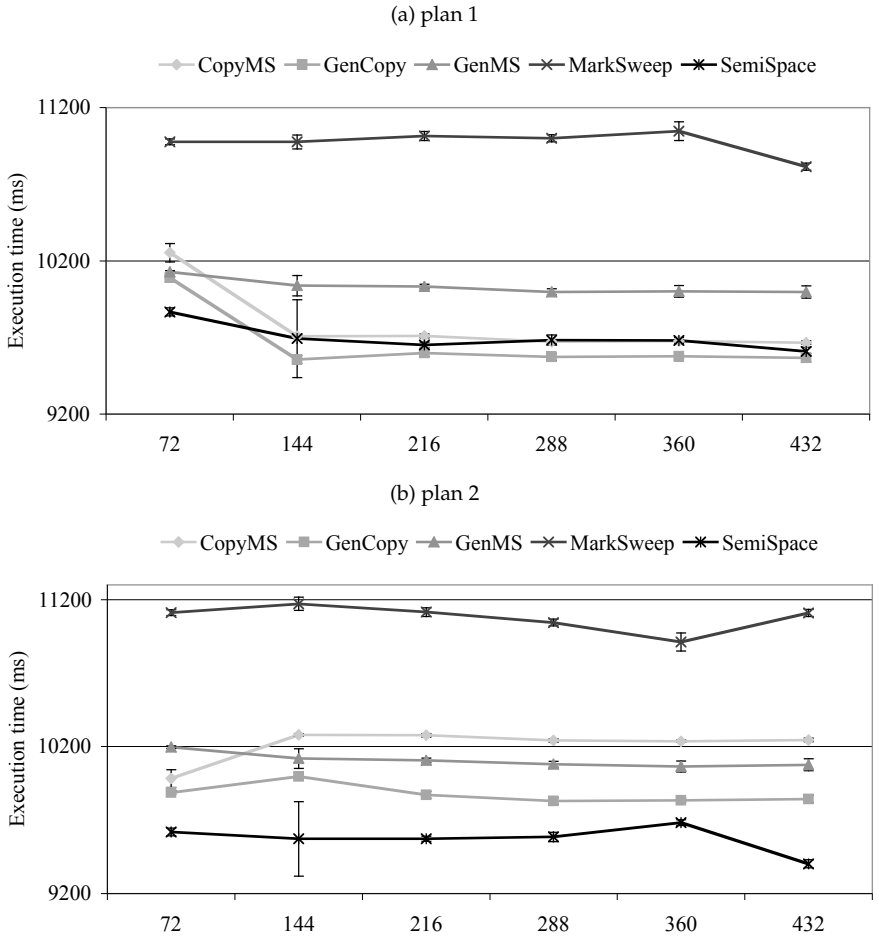


Figure 5.10: Comparison between the mutator execution times for *jython* using two different 10-iteration compilation plans as a function of the heap size for five garbage collectors. We show the mean of 10 measurements for each plan and the 95% confidence intervals.

10-iteration plans.

Figures 5.6, 5.7, 5.8 and 5.9 show the percentage of inconclusive and disagreeing comparisons for GC time and mutator time for SPECjvm98 and DaCapo, respectively, assuming stable replay and 10-iteration compilation plans. Although compilation plans mostly agree on the best GC strategy in terms of GC time (see Figures 5.6 and 5.7) – for some benchmarks, such as *jess*, *bloat*, *fop* and *mpegaudio*, all compilation plans agree – this is not the case for all benchmarks, see for example *antlr* in the 64MB to 96MB heap size range. In contrast to the large fraction of agrees in terms of GC time, this is not the case for muta-

tor time, see Figures 5.8 and 5.9. For some benchmarks, the fraction disagrees and inconclusives can be as large as 13% (*hsqldb*) and 35% (*fop*), respectively. (Again, *mpegaudio* is a special case for the same reason as above.)

To show that there are potentially very significant differences between compilation plans, we compare the mutator execution times of *jython* for each of the five garbage collectors in our experiment per heap size, for two different compilations plans obtained after running the benchmark for 10 iterations. Figure 5.10 shows that for the first plan, there is no clear winner given that there are very minimal difference between CopyMS, GenCopy and SemiSpace. However, the second plan paints a very different picture, where SemiSpace is clearly faster than all other collectors, the difference being over 3% for some heap sizes.

Analyzing mutator time.

The high fraction disagrees and inconclusives for mutator time in the above experiment raises an important question: is this observation a result of the effect that the GC strategy has on the data layout of the mutator, or in other words, is the GC strategy one of the main contributors to the high fraction disagrees and inconclusives? Or, is this observation simply a result of the performance variability observed across compilation plans and does the GC strategy not affect the mutator?

To answer this question, we employ a two-factor ANOVA with the two factors being the GC strategy and the compilation plan, respectively. The two-factor ANOVA then reports whether the variability in mutator time is due to the GC strategy, the compilation plan, their mutual interaction, or random noise in the measurements. In almost all cases, the garbage collector has a significant impact on mutator time at the 5% significance level. The same is true for both the compilation plans and the interaction between these two factors. Figures 5.11 and 5.12 show the percentage of the total variability in mutator time under stable replay that is accounted for by the garbage collector (SSA), the compilation plan (SSB), their interaction (SSAB), and the residual variability (SSE) due to random measurement noise for SPECjvm98 and DaCapo, respectively. Except for *mtrt*, garbage collection accounts for over 15% of the observed variability in these experiments. Remarkably, the GC strategy affects mutator time quite a lot for *mpegaudio*, accounting for over 50% of the observed variability, even though no time is spent in GC during the stable run. Figure 5.13 illustrates this further: it shows that the mutator time is affected by both the GC strategy and the compilation plan although there is no GC activity under the stable run. This is to be explained by the way the heap is laid out after the full GC between the mix and stable runs, i.e., the different GC strategies lead to a different heap layout for the stable run. Note that the fact that the 95% confidence intervals shown in the plot overlap, does *not* mean there is no significant difference between the sample sets.

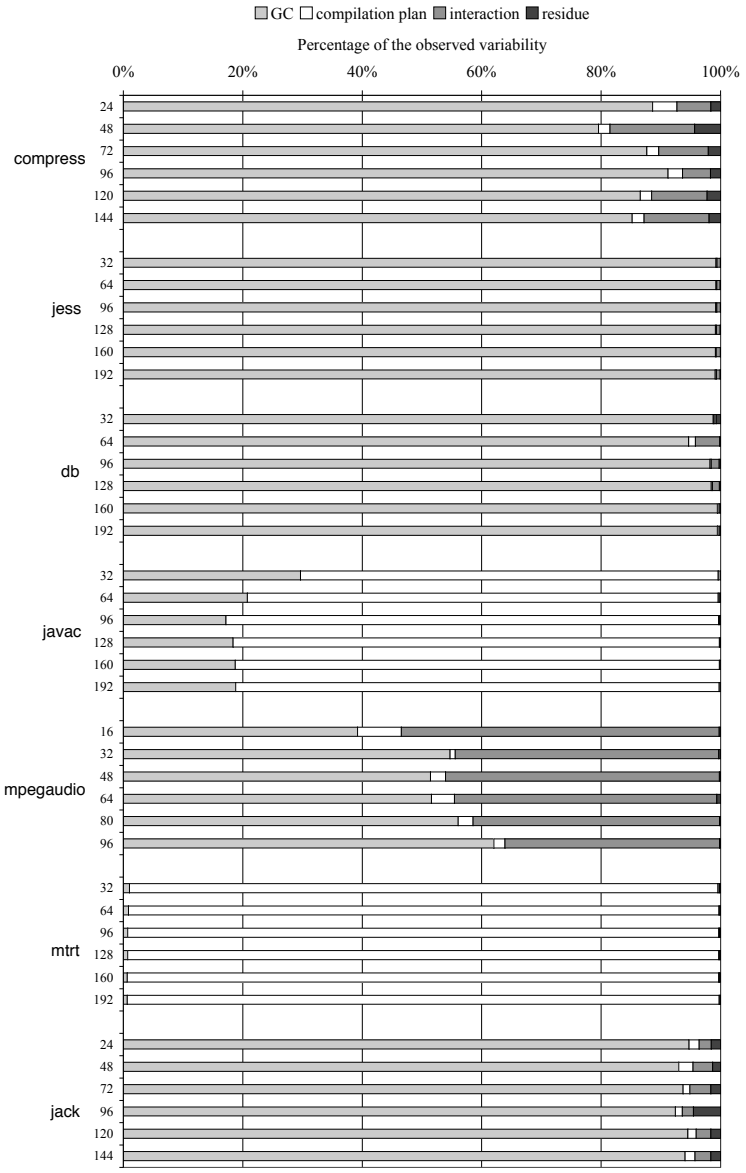


Figure 5.11: Percentage of the variability in mutator time accounted for by (i) the GC strategy, (ii) the compilation plan, (iii) the interaction between the GC strategy and the compilation plan, and (iv) the residual variability, for 10-iteration compilation plans on the Athlon XP under stable replay for SPECjvm98.

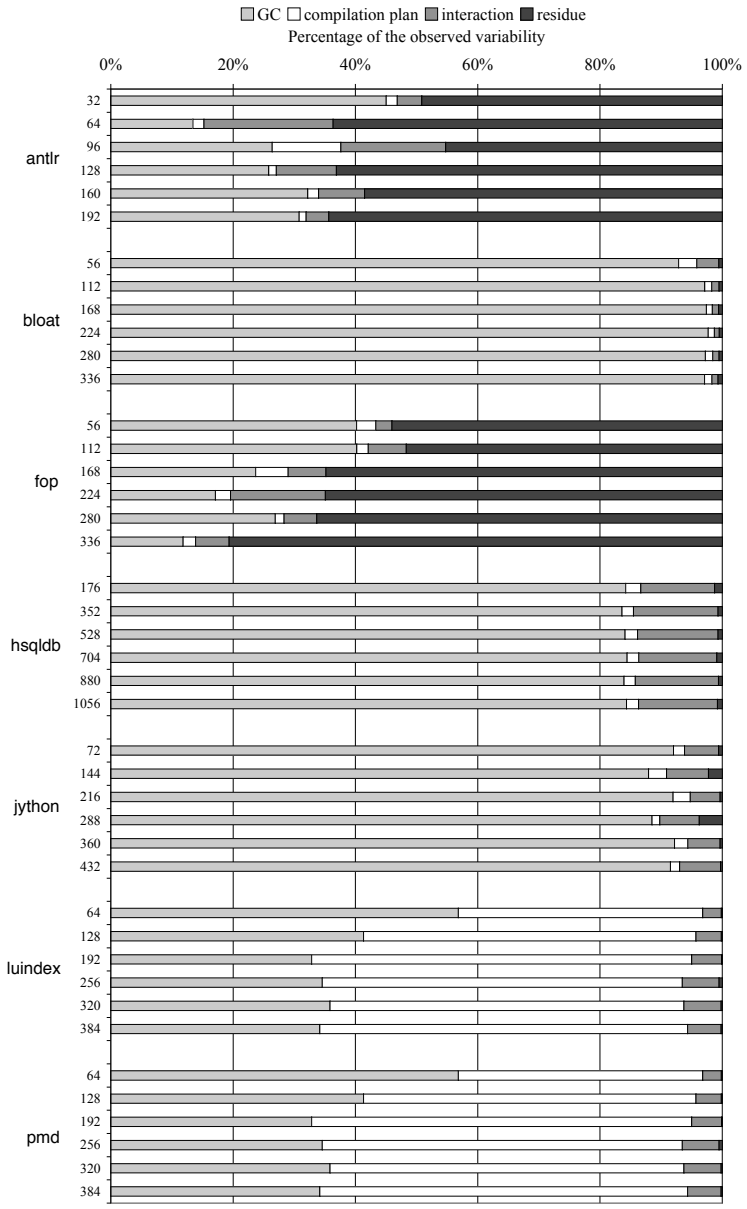


Figure 5.12: Percentage of the variability in mutator time accounted for by (i) the GC strategy, (ii) the compilation plan, (iii) the interaction between the GC strategy and the compilation plan, and (iv) residual variability, for 10-iteration compilation plans on the Athlon XP under stable replay for DaCapo.

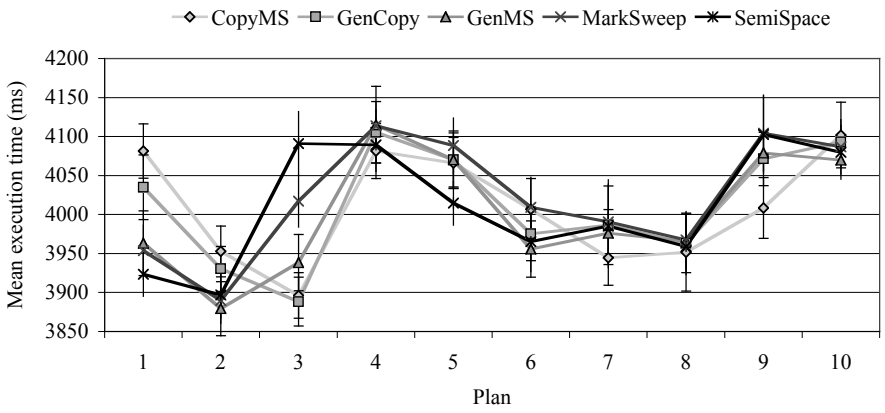


Figure 5.13: Interaction plot for *mpegaudio* with a heap size of 32MB. The reported values show the mean mutator time (in milliseconds) per compilation plan for different GC strategies on the AMD Athlon XP. Next to the mean values, we also show the 95% confidence intervals.

These results show that both factors, the GC strategy and the compilation plan, as well as their mutual interaction, have a significant impact on the observed variability. Most importantly for this study, we conclude that the GC strategy has a significant impact on the mutator time variability in this experiment, and thus the answer to the above question is that the large fraction of disagrees and inconclusives for mutator time is in part due to GC, and is not just a result of the variability observed across compilation plans.

5.3.4 Comparison with a majority plan

A so-called majority plan aims to capture information from multiple plans. The aim is to reduce the number of experiments that should be conducted, while simultaneously accounting for the differences in compilation decision that may be observed across plans. We determine each majority plan based on 10 individual plans. To compare both approaches, we use an ANOVA with a Tukey HSD post-hoc test in both the majority-plan and the multiple plans case, i.e., we simply aggregate the measurements obtained from using multiple plans during the ANOVA.

Figure 5.14 shows that in about 4% of the cases on average, majority and MP disagree under both mix and stable replay, i.e., they allow opposite conclusions to be drawn. In 16.8% and 12% of the cases for mix and stable replay respectively, the conclusion in inconclusive, i.e., one of the approaches claims there is no difference between the alternatives, whereas the other does find a significant difference.

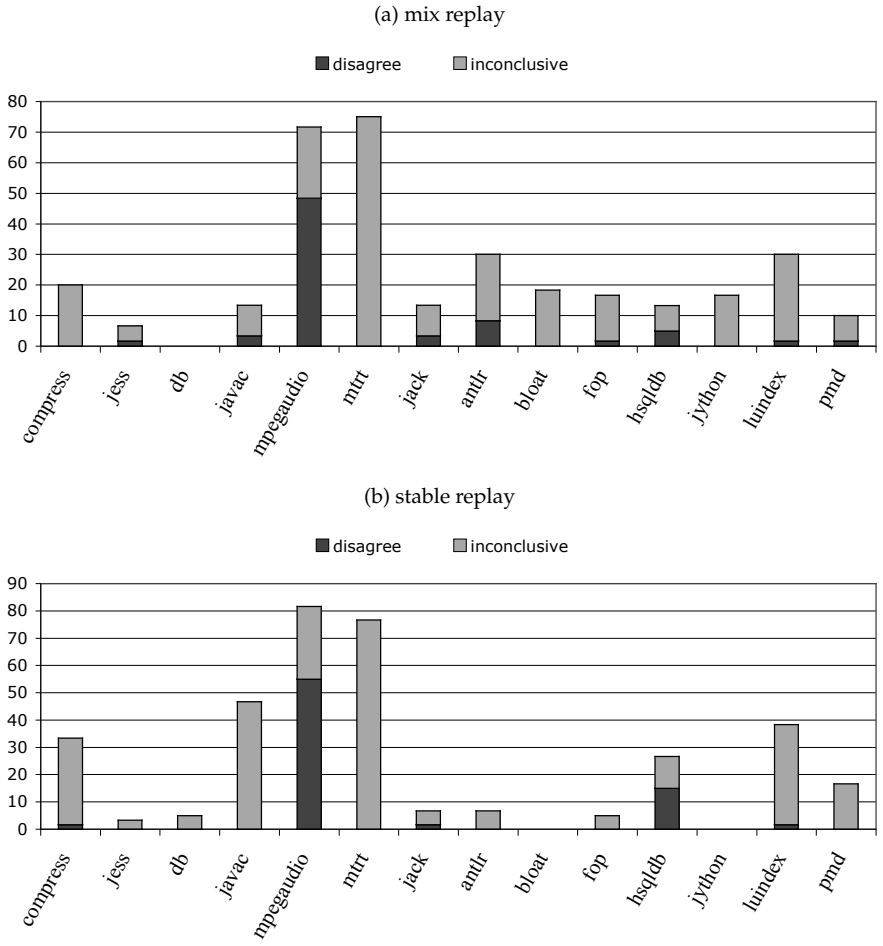


Figure 5.14: Percentage disagreeing and inconclusive comparisons under (a) mix replay, and (b) stable replay for all benchmarks when comparing majority plans versus multiple plans both with an ANOVA plus a Tukey HSD post-hoc test at a 5% significance level on the AMD Athlon XP.

5.3.5 Summary

As a summary from this section, we thus conclude that (i) different compilation plans can lead to execution time variability that is statistically significant; (ii) the reason for this runtime variability is the difference in the methods and their optimisation levels appearing in the compilation plans; and (iii) different compilation plans can lead to inconsistent conclusions in practical research studies. For these reasons we argue that, in order to yield more accurate performance results, replay compilation should consider multiple compilation plans instead of a single one at replay time as done in current replay compila-

tion practice.

5.4 Statistical analysis

Now that we have reached the conclusion that rigorous replay compilation should consider multiple compilation plans, we need statistically rigorous data analysis for taking statistically valid conclusions from these multiple compilation plans.

5.4.1 Multiple measurements

As mentioned before, the performance measurements for a given compilation plan are still subject to non-determinism. Therefore, it is important to apply rigorous data analysis when quantifying performance for a given compilation plan, as we have shown in the previous chapter. Before analysing the data in terms of whether an innovation improves performance, as will be explained in the following section, we first compute the average execution time per compilation plan.

Provided we have enough measurements per plan, the central limit theory applies and $\overline{x}$ is approximately Gaussian distributed. To reach the requirement of independence, we discard the first measurement for each compilation plan [50], i.e., the first virtual machine invocation as indicated in Section 4.3.

5.4.2 Matched-pair comparison

Comparing design alternatives and their relative performance differences is of high importance to research and development, more so than quantifying absolute performance for a single alternative. When comparing two alternatives, a distinction needs to be made between a experimental setup that involves corresponding measurements, versus a setup that involves non-corresponding measurements. Under replay compilation with multiple compilation plans there is an obvious pairing for the measurements per compilation plan. In particular, when evaluating the efficacy of a given innovation, the performance is quantified before the innovation as well as after the innovation, forming an obvious pair per compilation plan. This leads to a so-called *before-and-after* or *matched-pair* comparison [62, 71].

To determine whether there is a statistically significant difference between the means before and after the innovation, we must compute the confidence interval for the *mean of the differences* of the paired measurements. This is done as follows, assuming there are n compilation plans. Let $\bar{b}_j, 1 \leq j \leq n$, be the average execution time for compilation plan j *before* the innovation; likewise, let $\bar{a}_j, 1 \leq j \leq n$, be the average execution time for compilation plan j *after* the

innovation. We then need to compute the confidence interval for the mean $\overline{d}$ of the n difference values $\overline{d}_j = \overline{a}_j - \overline{b}_j, 1 \leq j \leq n$.

To compute the confidence interval, Equation 4.2 applies if the number of compilation plans is large (say, more than 30 [71]), otherwise Equation 4.3 applies, with the standard deviation s in this case computed as follows:

$$s = s_{\overline{d}} = \sqrt{\frac{\sum_{i=1}^{n}(\overline{d}_i - \overline{d})^2}{n-1}}.$$

Once the confidence interval is computed, we then verify whether the confidence interval includes 0. If this is the case, the null hypothesis H_0 that both alternatives have equal mean performance cannot be rejected. If the interval does not include 0, we must reject H_0 at a $1-\alpha$ confidence level. Again, there always remains a probability α that the observed differences are due to random effects, and that H_0 is erroneously rejected. As usual, we cannot assure with a 100% certainty that there is an actual difference between the compared alternatives.

5.5 Rigorous replay compilation

Figure 5.15 illustrates the overall replay compilation methodology that we advocate when comparing two alternatives. We start by collecting n compilation plans. For each of these compilation plans we then collect k performance numbers for both the 'before' and the 'after' experiments, and subsequently compute an average performance number per compilation plan before the innovation, $\bar{b}_j$, as well as after the innovation, $\bar{a}_j$. The differences between the alternatives per compilation plan, $\bar{d}_j = \bar{b}_j - \bar{a}_j$, then serve as input to the matched-pair comparison as explained in the previous section.

This replay methodology is more rigorous than current practice because it includes multiple compilation plans. The flip side though is that this methodology implies that more experiments need to be run. We now need to collect performance numbers for multiple compilation plans instead of a single compilation plan. This may be time-consuming and may be a concern under time pressure.

Fortunately, only a limited number of compilation plans need to considered. The reason is that matched-pair comparison leverages the likely observation that the variability in relative performance difference between the alternatives is smaller than the variability observed across the compilation plans. More precisely, as we observed in Section 5.3, the variability in performance between different compilation plans can be large. However, the intuition is that the variability in relative performance across alternatives *for a given compilation plan* is not very large. A compilation plan leading to high performance for one alternative is likely to also yield high performance for the other alternative, even if the absolute performance is different. In our experiments, we

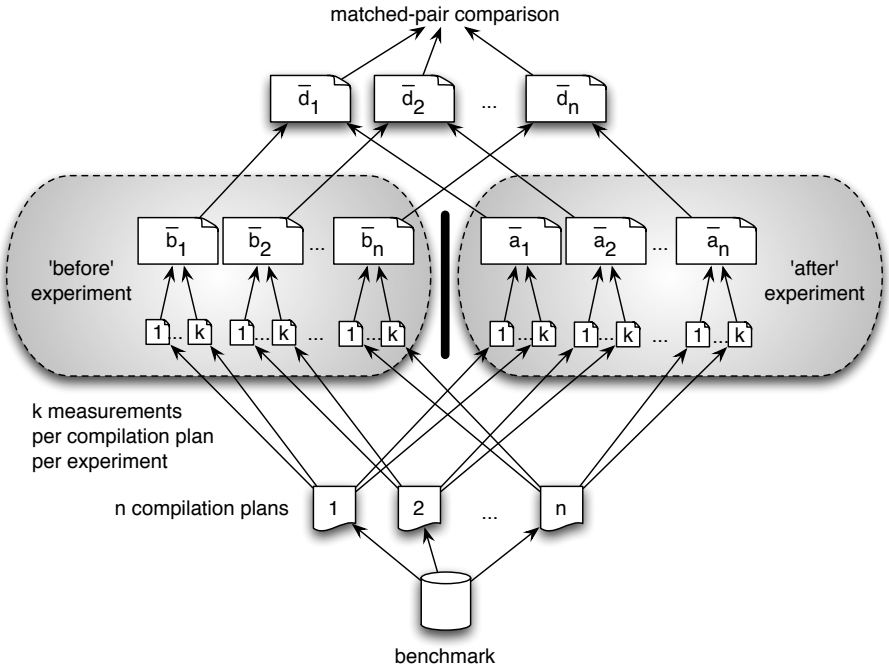


Figure 5.15: Replay compilation methodology using multiple compilation plans.

found this to be the case in general, as will be shown later. We will exploit this property to limit the number of compilation plans that need to be considered while maintaining a high accuracy and tight confidence intervals.

The underlying reason is that a matched-pair comparison exploits the property of paired or so-called corresponding measurements. To better understand this important property, we first need to explain how to compare two alternatives in case of non-corresponding measurements.

5.5.1 Non-corresponding measurements

Consider two alternatives and respective measurements $x_{1j}, 1 \leq j \leq n_1$ and $x_{2j}, 1 \leq j \leq n_2$; assume there is no correspondence or pairing. We now need to compute the confidence interval of the difference of the means. We first need to compute the averages $\bar{x}_1$ and $\bar{x}_2$ for the two alternatives. The difference of the means then is $\bar{x} = \bar{x}_2 - \bar{x}_1$. The standard deviation of the difference of the means then is

$$s_x = \sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}},$$

with s_1 and s_2 the standard deviation for the two respective alternatives.

We can now compute the confidence interval $[c_1, c_2]$ for the difference of the means, again based on the central limit theory:

$$c_1 = \bar{x} - z_{1-\alpha/2} \cdot s_x$$

$$c_2 = \bar{x} + z_{1-\alpha/2} \cdot s_x.$$

If the resulting confidence interval includes zero, we can conclude that, at the confidence level chosen, there is no significant difference between the two alternatives.

5.5.2 Comparison between corresponding and non-corresponding measurements

Let's now compare the confidence interval computed for corresponding measurements versus non-corresponding measurements. Assume $n_1 = n_2 = n$, and $b_i = x_{2i}$ and $a_i = x_{1i}$. Recall the confidence interval for corresponding measurements equals:

$$c_{1,2} = \bar{d} \pm z_{1-\alpha/2} \cdot \frac{s_{\bar{d}}}{\sqrt{n}},$$

or

$$c_{1,2} = \bar{d} \pm z_{1-\alpha/2} \cdot \sqrt{\frac{\sum_{i=1}^{n}(\bar{d_i} - \bar{d})^2}{n(n-1)}}.$$

For non-corresponding measurements, the confidence interval for the difference of the means equals:

$$c_{1,2} = \bar{d} \pm z_{1-\alpha/2} \cdot \sqrt{\frac{\sum_{i=1}^{n}(\bar{a}_i - \bar{a})^2}{n(n-1)} + \frac{\sum_{i=1}^{n}(\bar{b}_i - \bar{b})^2}{n(n-1)}}.$$

Comparing both confidence intervals boils down to comparing

$$\sum_{i=1}^{n}(\bar{d_i} - \bar{d})^2$$

for the corresponding measurements, versus

$$\sum_{i=1}^{n}(\bar{a}_i - \bar{a})^2 + \sum_{i=1}^{n}(\bar{b}_i - \bar{b})^2$$

for the non-corresponding measurements. Writing $\bar{d_i}$ as $\bar{b}_i - \bar{a}_i$, and $\bar{d}$ as $\bar{b} - \bar{a}$, enables expanding the expression for the corresponding measurements to:

$$\sum_{i=1}^{n}(\bar{a}_i - \bar{a})^2 + \sum_{i=1}^{n}(\bar{b}_i - \bar{b})^2 - 2 \cdot \sum_{i=1}^{n}(\bar{a}_i - \bar{a})(\bar{b}_i - \bar{b}).$$

By consequence, if the term

$$\sum_{i=1}^{n}(\bar{a}_i - \bar{a})(\bar{b}_i - \bar{b})$$

is positive, then the confidence interval for the corresponding measurements is smaller than the confidence interval for the non-corresponding measurements. In other words, corresponding measurements result in tighter confidence intervals if the performance variation is large across the compilation plans, i.e., $\bar{a}_i - \bar{a}$ and $\bar{b}_i - \bar{b}$ are large, and if the relative performance variation is limited across compilation plans when comparing two alternatives.

To illustrate this finding empirically through our GC case study, we compute the following ratio R:

$$R = \frac{\sum_{i=1}^{n}(\bar{d}_i - \bar{d})^2}{\sum_{i=1}^{n}(\bar{a}_i - \bar{a})^2 + \sum_{i=1}^{n}(\bar{b}_i - \bar{b})^2},$$

across all benchmarks and all heap sizes, for all pairwise GC strategy comparisons. If this ratio is smaller than one, this means that the confidence interval computed through matched-pair comparison is smaller than through statistics assuming non-corresponding measurements. Figure 5.16 shows the cumulative distribution of this ratio. The various graphs show that in the majority of the cases, matched-pair comparison indeed results in smaller confidence intervals of the difference of the means. For example, for DaCapo and stable replay, for over 85% of the cases, matched-pair comparison results in a smaller confidence interval.

5.5.3 Number of compilation plans

The above analysis shows that matched-pair comparison for analysing the performance results from multiple compilation plans is likely to result in tighter confidence intervals than using non-corresponding measurements statistics. This observation has an important implication. It means that for the same level of accuracy, i.e., for the same confidence interval size, fewer compilation plans need to be considered when using matched-pair comparison statistics instead of non-corresponding measurements statistics. Or, in other words, under matched-pair comparison, the number of compilation plans that are needed to obtain tight confidence intervals is limited.

We now leverage this observation to find a good trade-off between the number of compilation plans versus the number of measurements per plan to obtain accurate performance numbers. For exploring this trade-off, we again consider our GC case study in which we consider 5 GC strategies and 6 heap sizes. We now pairwise compare GC strategies per heap size through matched-pair comparison. Figures 5.17 and 5.18 show the fraction of inconclusive and disagreeing conclusions – averaged across all SPECjvm98 benchmarks in the top graph and averaged across all DaCapo benchmarks in the

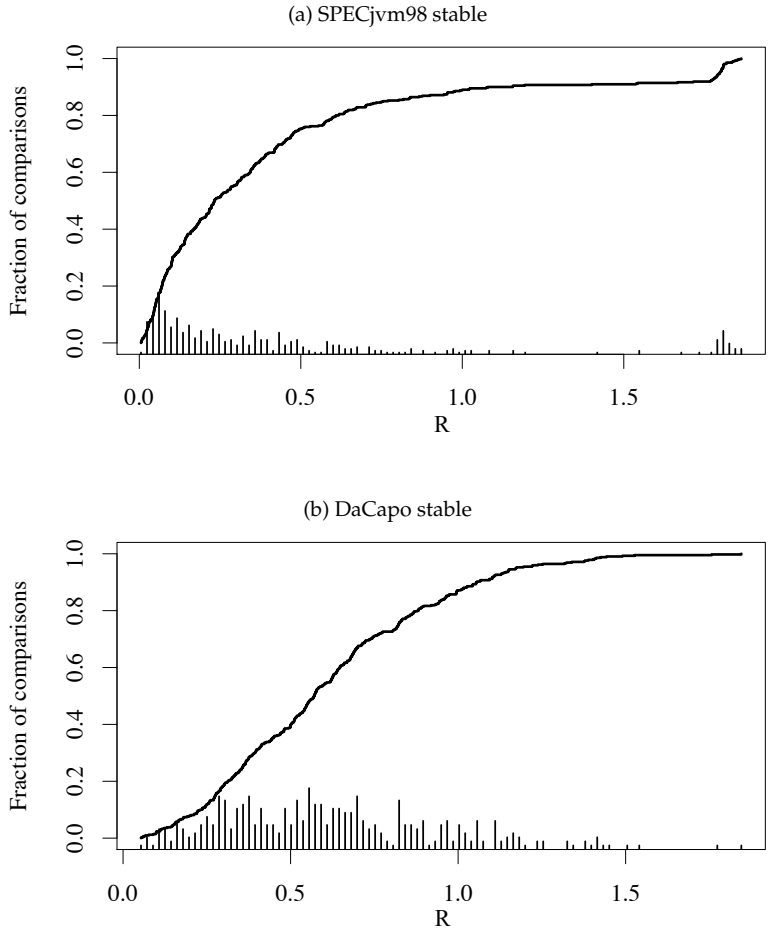
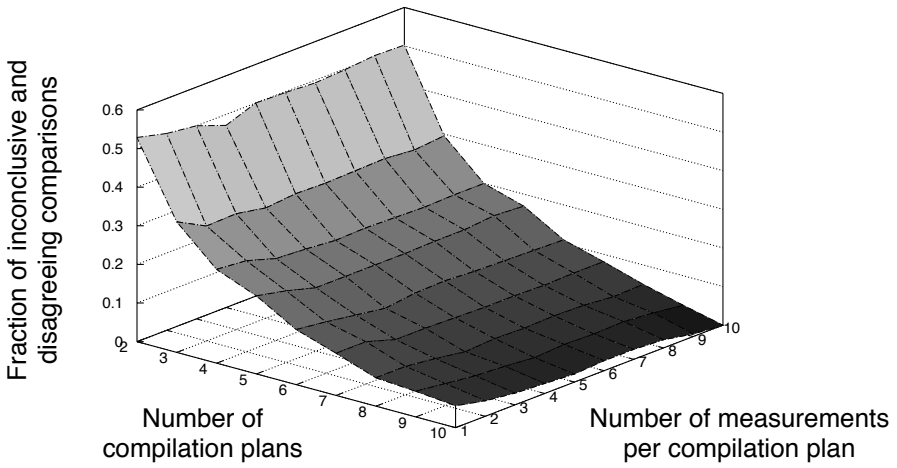


Figure 5.16: Cumulative distribution of the ratio R in confidence interval width between matched-pair comparison versus non-corresponding measurements statistics.

bottom graph – as a function of the number of compilation plans and the number of measurements per plan for stable replay for mutator and total execution time, respectively. The reference point per graph is the setup in which we consider 10 compilation plans and 10 runs per compilation plan. In other words, a point (x, y) in this graph shows the fraction inconclusive and disagree comparisons for x compilation plans and y measurements per plan compared to 10 compilation plans and 10 measurements per plan. We observe that the fraction inconclusive and disagree conclusions quickly decreases with even a limited number of, say 4 or 5, compilation plans. At the same time, the fraction inconclusive and disagree conclusions is fairly insensitive to the number

(a) SPECjvm98 stable mutator execution time



(b) DaCapo stable mutator execution time

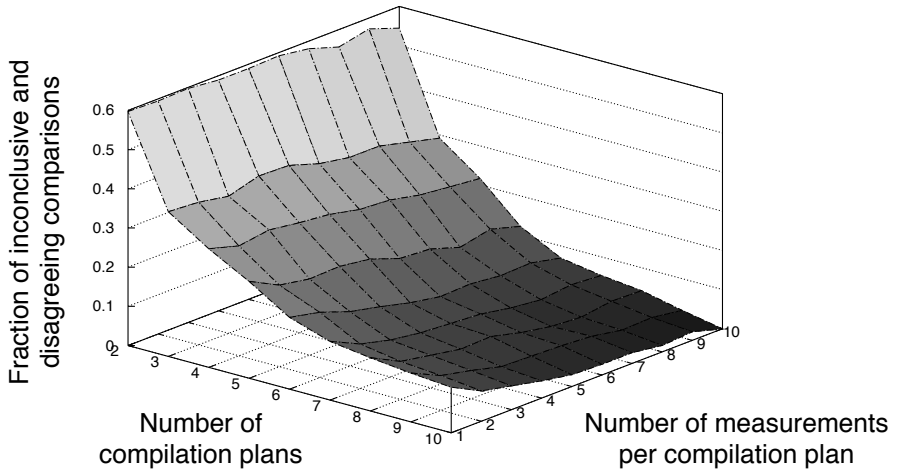
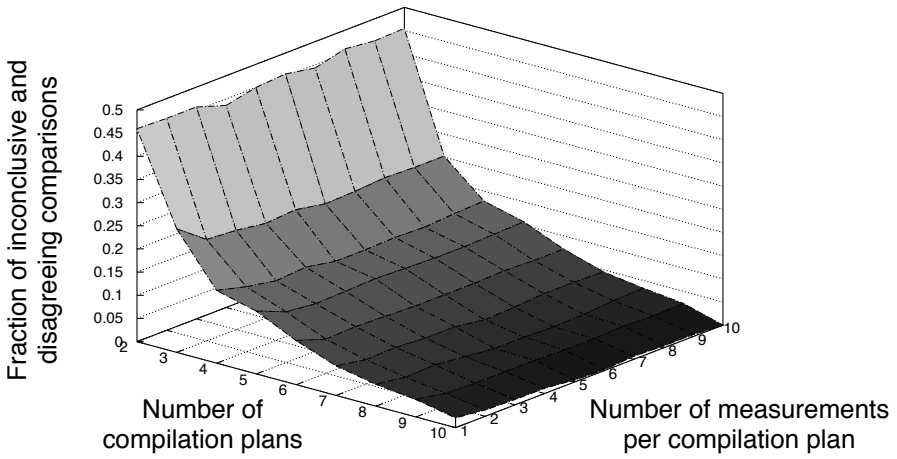


Figure 5.17: Exploring the trade-off between the number of compilation plans versus the number of measurements per 10-iteration compilation plan as measured on the AMD Athlon platform for mutator execution time.

(a) SPECjvm98 stable total execution time



(b) DaCapo stable total execution time

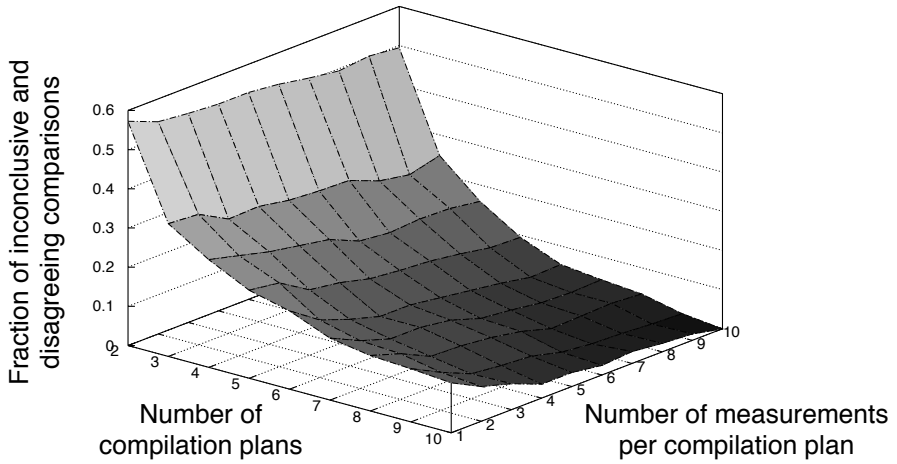


Figure 5.18: Exploring the trade-off between the number of compilation plans versus the number of measurements per 10-iteration compilation plan as measured on the AMD Athlon platform for total execution time.

of measurements per compilation plan. In other words, for a given experimentation time budget, it is more beneficial to consider multiple compilation plans rather than multiple measurements per compilation plan.

5.6 Conclusions

Replay compilation is an increasingly widely used experimental design methodology that aims at controlling the non-determinism in managed runtime environments such as the Java virtual machine. Replay compilation fixes the compilation load by inducing a pre-recorded compilation plan at replay time. The compilation plan eliminates the non-determinism due to timer-based sampling for JIT optimisation.

Current practice typically considers a single compilation plan at replay time, either selected out of a number of profiles, or obtained by combining various profiles. The key observation made in this chapter is that a single compilation plan at replay time does not account for the variability observed across different profiles, see Section 5.3. The reason is that different methods may be optimised at different levels of optimisation in different compilation plans. And this may lead to inconsistent conclusions across compilation plans in practical research studies. We have shown that majority is no surrogate for using multiple plans, see Section 5.3.4. We therefore advocate replay compilation using multiple compilation plans so that the performance number obtained from replay compilation is a better representative for average performance.

The statistical data analysis that we advocate under replay compilation with multiple compilation plans is matched-pair comparison. Matched-pair comparison considers the before and after experiments for a given compilation plan as a pair, and by doing so, achieves tighter confidence intervals in general than assuming unpaired measurements, as we have shown in Section 5.5.2. The reason is that replay compilation leverages the observation that the variability in the performance difference between two design alternatives is likely smaller than the variability across compilation plans. By consequence, replay compilation with multiple compilation plans and matched-pair comparison limits the number of compilation plans that need to be considered, and thus limits the experimentation time overhead associated with multiple compilation plans.

Finally, there is a difference between measurements obtained under replay compilation and measurements obtained during non-controlled execution. Consequently, we think it is best if both results are reported. At the end of the day, people are interested in global performance, even if they have chosen an alternative, e.g., a garbage collector, using replay compilation controlled runs.

Chapter 6

Performance? Dive into the application

Then there is the man who drowned crossing a stream with an average depth of six inches. – **W.I.E. Gate**

As we have showed in Chapter 2, the execution of a Java application involves a complex interaction between the Java code and the virtual machine (VM). Consequently, behaviour observed at the micro-architecture level is not only a function of the VM, or the application but includes a non-negligible fraction of behaviour caused by the interaction between these two aspects. If we take the continuous increasing size of applications and their complexity into account, it becomes clear that average performance numbers provide little insight into the application-specific behaviour. This pitfall can be avoided or circumvented by looking at method-level phase behaviour.

6.1 Introduction

Modern virtual machines are built from a large number of subsystems, such as the interpreter, compiler, optimiser, sampling system, thread scheduler, finaliser, garbage collector, etc. Hence, understanding behaviour in a more detailed manner than simply looking at average performance requires automated approaches to analyse Java workload behaviour.

In this chapter, we study method-level phase behaviour in Java workloads. We primarily focus on methods from the application itself. Secondary, we are interested in subsystems from the virtual machine, e.g., the garbage collector. For example, if the garbage collector exhibits worse performance than the

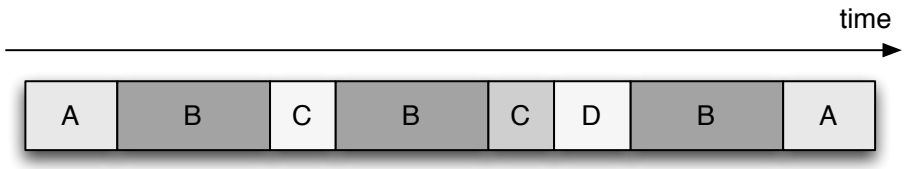


Figure 6.1: This part of the execution contains four phases (A, B, C, and D), three of which are recurring, whereas D occurs but once.

application methods, and thereby – depending on the amount of time spent in GC – degrading overall performance, the fastest and most viable option to improve performance may be to switch garbage collectors, if possible.

The notion of a phase is quite loosely defined in the literature. We use it in this dissertation in the following sense. A phase is defined as a set of segments of the program execution that exhibit similar behaviour but need not be temporally adjacent. This is illustrated in Figure 6.1. Here, we have four phases : A, B, C, and D. The first three occur more than once; D occurs but once. We say that phase A has multiple instances. Each of those instances are more similar to each other than to instances of the other phases, yet they need not be exactly the same, behaviour-wise.

The assumption underlying method-level phase behaviour, is that phases of execution correspond to the code that gets executed. In particular, different methods are likely to result in dissimilar behaviour, whereas separate invocations of the same methods plausibly result in similar behaviour. Several studies have checked whether methods, or functions, have the appropriate granularity to allow detection of program phase behaviour [9, 60, 66]. They indicate that the method level is at least as good as more detailed levels, such as the loop level or the basic block level. This is especially the case for applications with a lot of method calls where each method is quite small [66], such as Java [88]. This fits our purpose very well, because methods are sufficiently course-grained to allow identifying major phases, yet at the same time they are fine-grained enough to provide the desired detail.

To avoid the pitfall, i.e., average performance does not provide enough information, we use an off-line analysis – the method-level phase behaviour analysis – that consists of three steps.

1. We determine how much time the Java workload spends in different methods of the application. This is done by instrumenting all methods to read microprocessor performance counter values to track the amount of time that is spent in each method. The result of this first step is an annotated dynamic call graph.

2. Using an offline analysis we determine the methods in which the application spends a significant portion of its total execution time with the

additional constraint that one invocation of the method takes a significant portion of the total execution time as well. This is to avoid selecting methods that are too small, e.g., getters and setters. Instrumenting such methods would produce too much overhead and cause too much perturbation by the instrumentation. Additionally, adding instrumentation might interfere with the inlining system, especially on small methods.

3. During a second run of the application, these selected methods are instrumented and performance characteristics are measured using hardware performance counters. We are particularly interested in a number of characteristics such as branch misprediction rate, cache miss rate, number of retired instructions per cycle, etc. This results in detailed performance characteristics for the major method-level execution phases of the Java application. In addition to the method-level phases, we also measure performance characteristics for major parts of the VM, such as the compiler/optimiser, the garbage collector, the class loader, the finaliser, etc.

There are several interesting applications for this work. First, it provides application programmers insight in the behaviour of the Java workload in all its complexity so they can optimise its performance. Obviously, average performance numbers can only say if a change improves performance, but cannot point out what to change. Using automatic techniques to characterise Java workloads can thus be helpful to identify performance bottlenecks with limited effort. Second, for VM developers, automatic workload characterisation helps to get insight in how a Java application interacts with its VM, which allows improving the performance of the VM under development. For example, if our technique indicates that the branch prediction accuracy of a certain method falls way short of the mark, the VM developer can see if the code generated by his JIT contributes to the problem – and fix it. Third, our approach also provides interesting insights into phase behaviour. Detecting program execution phases and exploiting them has received increased attention in recent literature. Various authors have proposed ways of exploiting phase behaviour. One example is to adapt the available hardware resources to reduce energy consumption while sustaining the same performance [9, 37, 60, 98]. Another example is to use phase information to guide simulation-driven processor design [97]. The idea is to select one single sample from each phase for simulation instead of the complete benchmark execution. On the software side, JIT compilers in VMs [4, 5] and dynamic optimisation frameworks [8, 80] heavily rely on implicit phase behaviour to optimise code. Gu and Verbrugge [52] use phase behaviour to optimise adaptive recompilation, which results in an average speedup of 4.5%. They use hardware performance counter event to detect phases where they rely on the event density to find phase transitions.

This chapter is organised as follows. The next section details on our experimental setup. Section 6.3 discusses our off-line approach for identifying

Mnemonic	Description
cycles	elapsed clock cycles during execution
ret_instr	retired instructions
L1-D-misses	L1 data cache misses
L2-D-misses	L2 data cache misses
L1-I-misses	L1 instruction cache misses
L2-I-misses	L2 instruction cache misses
L1-L-misses	L1 cache load misses
L1-S-misses	L1 cache store misses
L2-L-misses	L2 load misses
L2-S-misses	L2 store misses
I-TLB-misses	Instruction TLB misses
D-TLB-misses	Data TLB misses
br_mpred	branches mispredicted
res_stall	resource stalls

Table 6.1: Performance counter events traced on the AMD Athlon XP.

method-level phase behaviour. Section 6.4 discusses the statistical data analysis techniques that we have used to quantify the variability between and within phases. The results of our phase analysis are presented in Section 6.5. Section 6.6 discusses related work. Finally, we conclude in Section 6.7.

6.2 Experimental setup

The experiments described in this chapter were all performed on a single platform, i.e., an AMD Athlon XP[1], see also Appendix A. Once more, we use the hardware performance counters available on the processor to gain insight into the behaviour of the Java applications at the method level. Recall that the Athlon XP has four performance counter registers, each of which can be used for counting any performance event. In this study, we trace the 14 events that are listed in Table 6.1. These events are commonly used in architectural studies to analyse program behaviour. However, just as we did in Chapter 2, we use the normalised events with respect to the number of retired instructions, e.g., the CPI, the number of branch misses per instruction, etc. We will further refer to them as performance characteristics. Once again, we need to perform several runs before we can gather all the information from Table 6.1.

To ease the measurement, we use the Performance API [27] (shortened to PAPI), as an extra layer between the native *perfctr* module in the kernel and the virtual machine in which we gather the event counts. PAPI is a high-level li-

[1]The processor we use in this chapter is clocked at 1.33Ghz, has but 256KB L2 cache and was fitted with 1GB of memory.

brary presenting a uniform interface to the performance counters on multiple platforms. The *perfctr* kernel patch allows tracing a single process, maintaining the state of the performance counters across kernel thread switches. The PAPI library presents a uniform manner for accessing the performance counters through the kernel module. Not all PAPI defined events are available on every platform, and not all native AMD events can be accessed through PAPI. However, for our purposes, it provides a sufficient set of events.

To atone for the non-determinism in the hardware performance counter data, we collect four measurements and use the mean value during the analysis.

The experiments have all been conducted with the Jikes RVM [3, 4, 29]. Unlike the experiments described in the rest of this dissertation we use the CVS version from January 2004 in this chapter. We use the adaptive compilation strategy, together with the CopyMS[2] garbage collector. This is a semi-generational collector with a copying nursery and mark-sweep mature space. However, there is no write barrier, and no remembered set, and every collection processes the entire heap. Nursery survivors are copied into the mature space and the old space is collected using a mark-sweep algorithm. Hence, the collector is a full heap collector. This also accounts for the fact that quite some time is spent in GC, unlike with, e.g., the GenMS collector.

The threading system multiplexes n Java threads (application and VM) onto m native (kernel) threads that are scheduled by the operating system. A command line option specifies the number of kernel threads that are created by the Jikes RVM. Usually, there is one kernel thread used for each physical processor, also referred to as a virtual processor because multiple Java threads can be scheduled by the VM within the single kernel thread. In our setup, we have used a single virtual processor.

The Jikes RVM implementation we use for the experiments in this chapter has support for accessing the performance counters through PAPI. The HPM subsystem of Jikes is responsible for providing access to the PAPI library, for example, for starting, stopping and resuming the counting of events, and for reading the actual counter values. Figure 6.2 illustrates the scheme we use. The standard implementation in Jikes does the following. Each VM thread – recall that a Java thread is mapped onto a virtual machine thread – keeps track of the counter events it causes while it is executing on Jikes' virtual processor. At each virtual context switch, the removed thread reads the counter values, accumulates them with its old values and restarts the counters. Consequently, a thread only observes performance events that occur while it is executing. This mechanism for reading performance counter values is the standard implementation within the Jikes RVM. For a more detailed description on this, we refer to [108]. In Section 6.3, we will detail how we extended this approach for measuring performance counter values on a per-method basis.

It would be hard to conduct experiments without benchmarks. In this

[2] At the time when the experiments were conducted, this was the default collector.

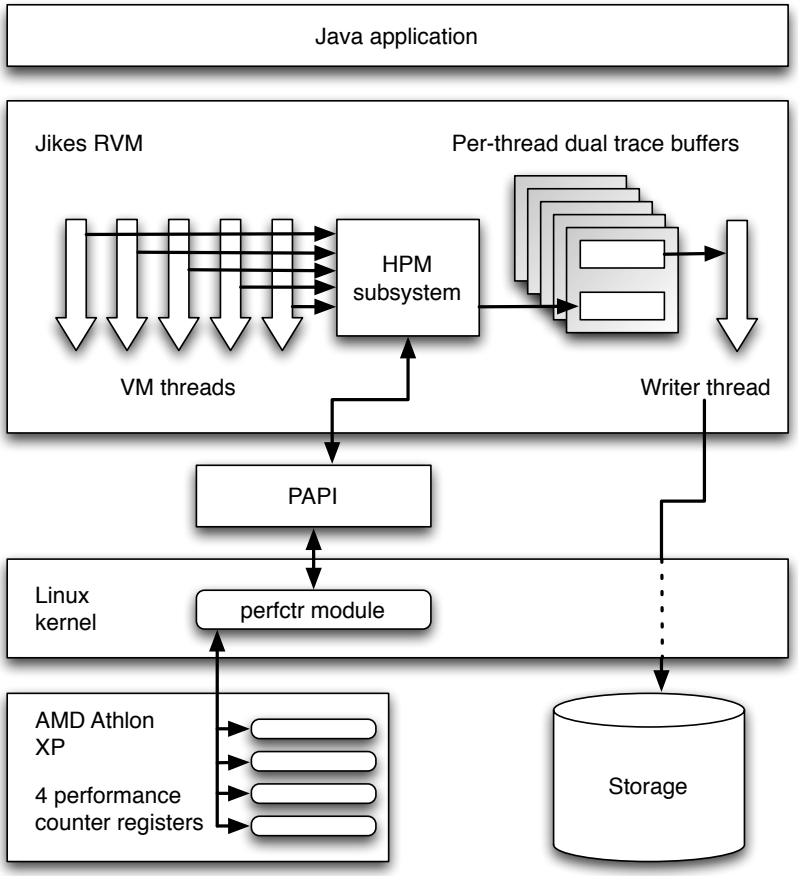


Figure 6.2: Overview of the Jikes RVM tracing system. The write thread is a separate POSIX/OS thread, such that the Jikes virtual processor is not blocked while trace data is stored to disk.

chapter we focus on the SPECjvm98 suite and SPECjbb2000. We use all SPECjvm98 benchmarks and we execute them using the *s100* input set. For each benchmark, we use a fixed heap size of 64 MB. SPECjbb2000 [102] is a server-side benchmark suite focusing on the middle-tier, the business logic, of a three-tier system. We have used a modified version of this benchmark, known as *PseudoJBB*, which executes a fixed amount of transactions, instead of running for a predetermined period of time. The benchmark was run with 8 warehouses. The heap size was set to 384 MB.

Remark 6.1. *The experiments described in this chapter were conducted prior to the findings described in Chapters 4 and 5. While the analysis presented here is sound and statistically rigorous, we did not use as many samples as we advised earlier, lim-*

iting ourselves to four samples for each configuration. We have used a fixed heap size (64MB for SPECjvm98 and 384MB for PseudoJBB). If we would redo the experiments, we would adopt a heap size strategy used in the previous chapters, and we would use multiple garbage collection algorithms.

For our current purpose, it suffices to have four samples. The bottom line is that we want to find out what the bottlenecks are in our application, by zooming into the execution, rather than by using global performance numbers. For this approach to be useful to a programmer, it should require but a few executions of the application. Of course, if better precision is desired, more than four runs should be used.

6.3 Method-level phases

As mentioned in the introduction, the method-level phase characterisation proposed in this dissertation is an off-line technique which requires two runs of the same application and which correlates performance characteristics with the method-level phases. This is particularly useful for Java and VM developers during performance analysis of their software. Method-level phase behaviour allows them to improve their software since the performance characteristics that are obtained from the hardware performance monitors are linked directly to the source code of the application and virtual machine. In comparison, Sweeney et al. [108] read performance counter values on virtual context switches, and output the method ID of the method that is executing at that time to a trace file – theirs is an online tool. In our methodology, we specifically link performance counter values to the methods. This is an important difference because the method executing at the virtual context switch is not necessarily the method that was executed during a major fraction of the time slice just before the virtual context switch. A second motivation for studying off-line phase behaviour is that it can be used as a reference for dynamic (online) phase classification approaches. In other words, the statically identified phases can be used for evaluation purposes of dynamic phase classification techniques. Obviously, to identify recurring phases, static phase analysis has the advantage over dynamic phase analysis as it can look at the 'future' by looking ahead in the trace file. A dynamic approach on the other hand has to anticipate phase behaviour and as such, can result in suboptimal phase identification. In addition, the resources that are available during off-line analysis are much larger than in case of on-line analysis, irrespective whether phase classification is done in software or in hardware. Yet another motivation for studying off-line method-level phase classification is for embedded and specialised environments in which the a priori information concerning the phase behaviour in the Java application can be useful.

The following issues are some of the more specific goals we want to meet using our off-line phase analysis approach.

- We want to gather information from the start to the end of the program's

execution. We want maximal coverage with no gaps, i.e., no part of the execution should be unmonitored.

- The overhead when profiling methods should be small enough not to interfere with normal program execution. This means that tracing all executed methods is not a viable option.

- We want to gather as much information as possible. At a minimum, the collected information should be sufficiently fine-grained such that transitions in Java performance characteristics can readily be identified. Such transitions can be caused by thread switches, e.g., the garbage collector is activated, or because the application enters a method that shows different behaviour from previously executed methods.

To meet these goals, we use the following off-line phase analysis methodology. During a first run of the Java application, we measure the number of elapsed clock cycles in each method execution. This information is collected in a trace file that is subsequently used to annotate a dynamic call graph. A dynamic call graph is a tree that shows the various method invocations during a program execution when traversed in depth-first order [2]. In a second step of our methodology, we use an off-line tool that analyses this annotated dynamic call graph and determines the major phases of execution. The output of this second step is a Java class file that describes which methods are responsible for the major execution phases of the Java application. In the third (and last) step, we link this class file to the VM and execute the Java application once again. The Java class file that is linked to the VM forces the VM to measure performance characteristics using the hardware performance monitors for the selected methods. The result of this run is a set of detailed performance characteristics for each method-level phase.

6.3.1 Mechanism

This section briefly discusses how a Java workload is profiled in our methodology. This instrumentation mechanism is used for both the initial profiling (step 1) and the gathering of the performance characteristics (step 2). To make this possible, we instrument the application methods when they are compiled by one of Jikes' compilers[3]. Given that a method consists of three main parts – (i) a prologue, (ii) a body, and (iii) an epilogue – the instrumentation code is added in the prologue and at the beginning of the epilogue.

Figure 6.3 illustrates how the actual tracing is done. First, to avoid counting events in the instrumentation code, the PAPI interface is asked to stop the counters. Subsequently the performance counters values are read, and stored in the buffer corresponding to the thread which is executing. As soon as the

[3]We do not consider wrapper methods, to avoid these being chosen as the starting point of a phase.

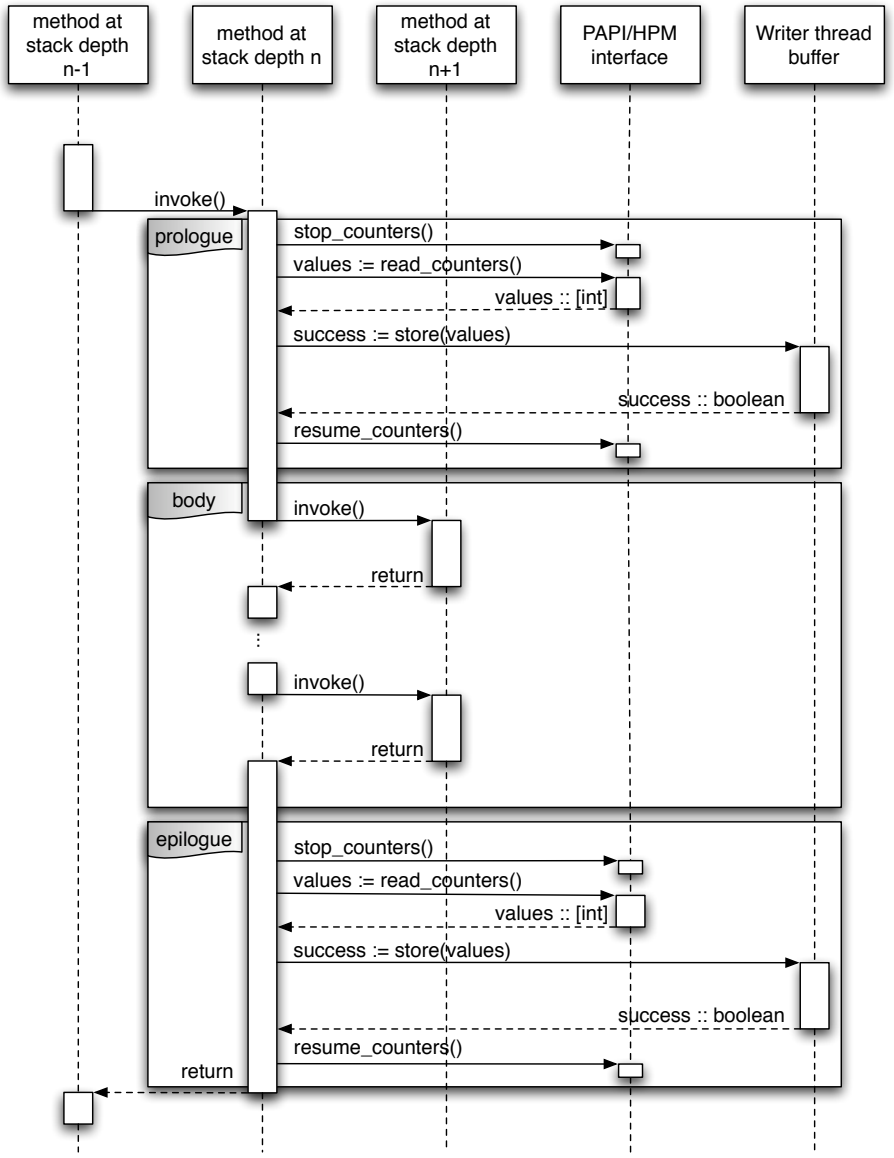


Figure 6.3: Tracing the performance counter events at the prologue and epilogue of a method at stack depth n.

values as stored, counting is resumed. We take care to log the correct values even in the event of a method being interrupted by an exception, by instrumenting Jikes' exception handling mechanism.

The data is stored in two trace files: (i) information pertaining to the compiler activity, i.e., method IDs and compilation level, and (ii) the actual counter values, stored in execution order, tagged with a thread ID.

Next to the application methods, we also log information about the virtual machine components, such as the garbage collector, compilers and finaliser. For specific details regarding the instrumentation, we refer to our OOPSLA 2004 publication [49].

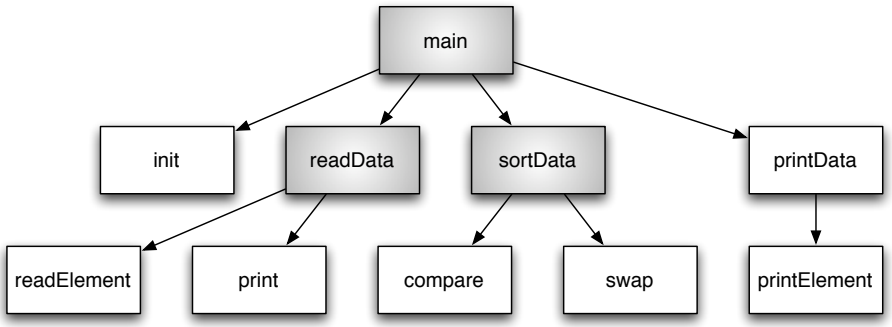
6.3.2 Phase identification

This section discusses how we identify phases using our off-line phase identification tool. Our tool takes a trace file with timing information about method calls and thread switches – as briefly explained in the previous section – as input, analyses it, and outputs a list of unique method names that represent the phases of the application.

To select method-level phases, we use the algorithm proposed by Huang et al. [60] which requires two parameters, called θ_{weight} and θ_{grain}. The algorithm essentially selects methods taking two conditions into account: (i) the total fraction of the execution time spent in the method (i.e., across all its instances) is sufficiently large, and (ii) each invocation of said method lasts long enough. The former – realised through θ_{weight} – assures that only important (hot) methods are selected, the latter – realised through θ_{grain} – assures that no methods are selected that are too short, e.g., getters and setters. The next example describes this process in detail.

Example 6.1. *To illustrate the phase identification algorithm, consider the call graph in Figure 6.4. It depicts a call tree that is the result of analysing the trace file of a fictitious sort program. Note that each of the nodes in the tree may appear more than once in the execution. The sort program reads the data to be sorted, prints an intermediate status message to the screen, sorts the data, and finally prints the sorted data before terminating. For simplicity, abstract time units are used. The table in Figure 6.4 also shows the total time spent in each method, as well as the time spent per invocation.*

To identify program phases, our tool first computes the total and average execution times spent in each method. For all methods, these times include the time spent in their callees. In order for a method to be selected as a program phase, its total execution time needs to be at least a fraction θ_{weight} of the program's total execution time, and the average execution time should take at least a fraction θ_{grain} of the program's total execution time on average. In our running example, $\theta_{\text{weight}} = 10\%$ and $\theta_{\text{grain}} = 5\%$, would select methods whose total execution time is more than 180 and whose average execution time is more than 90 — `main`, `readData` *and* `sortData`, *respectively.*



method	information		
name	total time	time/call	calls
main	1800	1800	1
init	30	30	1
readData	300	200	1
readElement	200	4	50
print	30	30	1
sortData	1300	1300	1
compare	600	2	300
swap	500	2	250
printData	170	170	1
printElement	150	3	50

Figure 6.4: Phase identification example.

6.4 Statistical techniques

To quantify the variability within phases, we use the Coefficient of Variation (CoV). We first measure the CoV for a given performance metric within a phase. This is defined as the standard deviation divided by the average value for that metric within the given phase. The overall CoV is then obtained by averaging over the various phases after weighting with the execution time spent in each of the phases. The smaller the CoV, the less variability is observed within a phase.

Once more, we use ANOVA [62] to quantify the variability within the phases versus the variability between the phases. The alternatives here are the various phases. Equations 4.5 through 4.9 are used to decide if the null-hypothesis that all phases have equal mean performance numbers can be rejected. The p-value corresponding to Equation 4.8 will be smaller than 0.05 for a 95% level of significance or smaller than 0.01 for a 99% level of significance.

6.5 Results

In this section we describe the results of our experiments. We thoroughly check if the selected method-level phases match our criterion that there is sufficient difference between phases. We then look at the behaviour of virtual machine components versus the application phases and present an analysis of the method-level phases, showing that performance bottlenecks can be located.

6.5.1 Identifying method-level phases

While we want to get a closer look at the per-method performance, tracing all methods at their prologue and epilogue is far too intrusive. Hence, we first determine a set of method-level phases such that the incurred overhead is relatively low, and such that we still get a sufficiently detailed picture of performance at the level of the executed methods. We use the values θ_{weight} and θ_{grain} for this purpose. The chosen values depend on three parameters: (i) the maximum acceptable overhead, (ii) the required level of information, and (iii) the application itself. Our off-line tool provides an estimate for both the information yielded per possible (θ_{weight} , θ_{grain}) pair and its estimated overhead.

These estimates are illustrated in Figures 6.5 and 6.6, showing results for *jack* and *pseudoJBB*, respectively. In each figure, the top graph presents the number of selected method-level phases as a function of both θ_{weight} and θ_{grain} . The bottom graphs shows the estimated overhead based on the same parameters. The latter is computed as the number of profiled methods divided by the total number of method invocations. Note that this is not the same as coverage, since selected methods also include their callees – the coverage is always 100% in our methodology. For the other benchmarks we obtained similar results. Clearly, the larger θ_{weight} , the fewer methods will be selected. The higher the value of θ_{grain} , the fewer short-running methods will be selected.

Using the plots in Figures 6.5 and 6.6, we choose appropriate values for θ_{weight} and θ_{grain} for each benchmark by inspecting the curves with the estimated overhead. We choose θ_{weight} and θ_{grain} in such a way that the estimated overhead is smaller than 1%, i.e., we want less than 1% of all method invocations to be instrumented. The results for each benchmark are shown in Table 6.2. Note that the user has some flexibility for determining appropriate values for θ_{weight} and θ_{grain} ; this allows the user to determine the number of selected method-level phases according to his interest.

So far, we have used an estimated overhead which is defined as the number of profiled method invocations versus the total number of method invocations of the complete program execution. To validate these estimated overheads, i.e., to compare with the actual overheads, we proceed as follows. We measure the total execution time of each benchmark (without any profiling

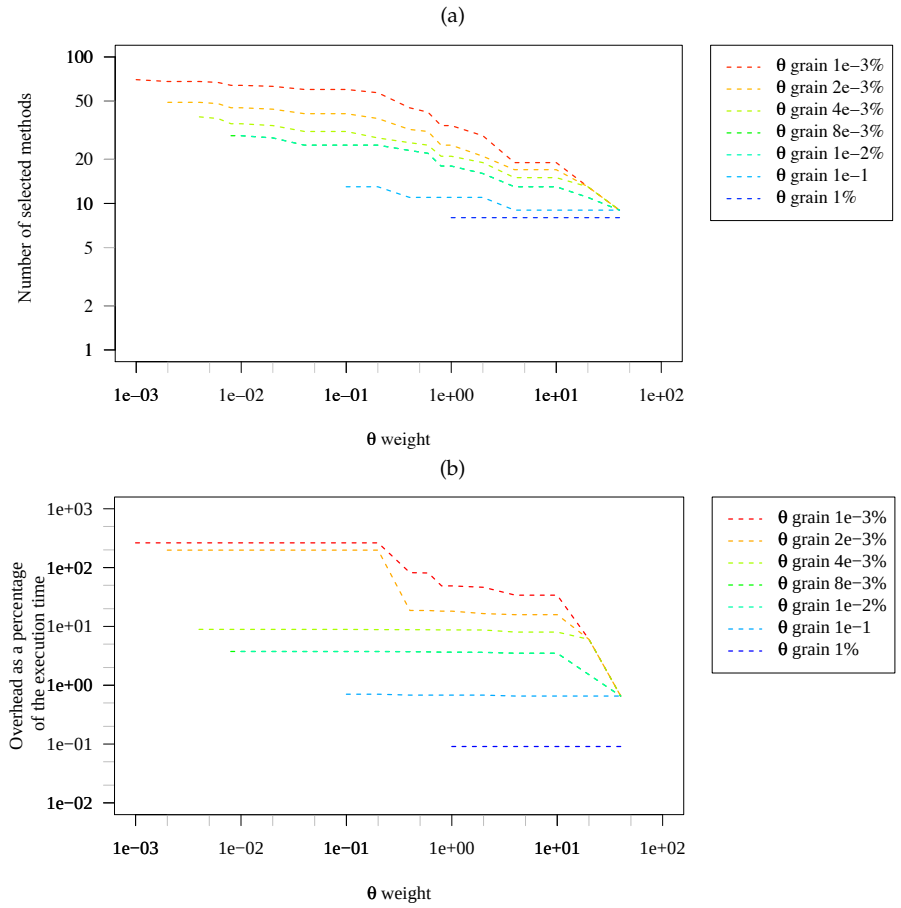


Figure 6.5: Estimating the overhead as a function of θ_{weight} and θ_{grain} for *jack*. The top graph shows the number of selected method-level phases; the bottom graph shows the estimated overhead.

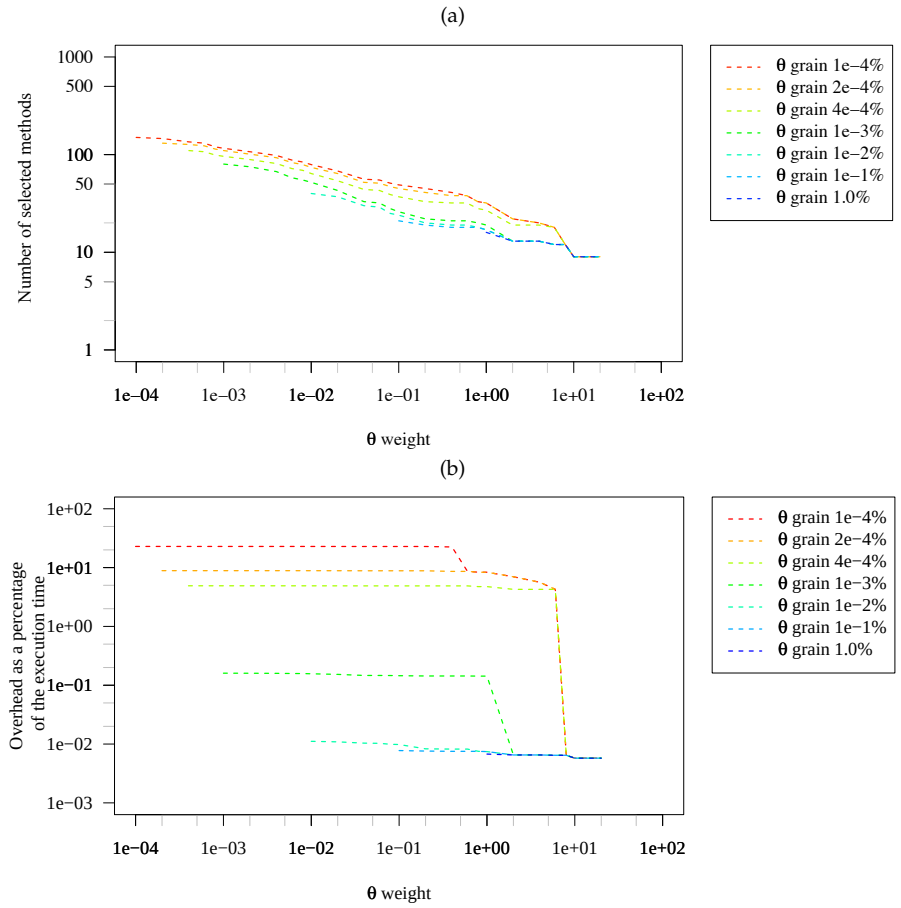


Figure 6.6: Estimating the overhead as a function of θ_{weight} and θ_{grain} for *pseudoJBB*. The top graph shows the number of selected method-level phases; the bottom graph shows the estimated overhead.

Benchmark	Configuration		Overhead (%)		Number of phases (total)	
	θ_{weight} (%)	θ_{grain} (%)	estimated	real	static	dynamic
compress	8×10^{-6}	6×10^{-6}	1.84	1.82	49 (54)	2,664 (19,726,311)
jess	1.0	1.0	1.22	1.27	10 (211)	23 (22,693,249)
db	8×10^{-6}	6×10^{-6}	7.17	5.61	52 (57)	32,223 (1,484,605)
javac	2×10^{-2}	6×10^{-3}	2.61	2.11	29 (503)	9,864 (23,388,699)
mpegaudio	2×10^{-2}	2×10^{-3}	10.75	3.52	23 (191)	40,064 (29,338,068)
mtrt	10^{-2}	10^{-3}	24.68	7.83	30 (94)	88,719 (14,859,306)
jack	1.0	10^{-2}	3.98	4.28	18 (182)	2,528 (4,292,580)
PseudoJBB	2×10^{-1}	2×10^{-4}	3.69	6.65	52 (381)	29,599 (16,224,804)

Table 6.2: Summary of the selected method-level phases for the chosen θ_{weight} and θ_{grain} values: overhead (estimated versus real), the number of static and dynamic phases.

enabled) and compare this with the total execution time when profiling is enabled for the selected methods. The actual overhead is defined as the increase in execution time due to adding profiling. Measuring the wall clock execution time is done using the GNU/Linux `time` command. Table 6.2 compares the estimated overhead and the actual overhead. We observe from these results that the actual overhead is usually quite small and mostly tracks the estimated overhead quite well. This is important since determining the estimated overhead is more convenient than measuring the actual overhead. In two cases, the estimate is significantly larger than the measured overhead, i.e. for *mpegaudio* and for *mtrt*. This may be due to the simple estimation formula. However, the formula yields a *safe* estimation, i.e., it overestimates the actual overhead in most cases.

For completeness, Table 6.2 also presents the number of unique method-level phases – the number of times a method-level phase is seen at least once – as well as the number of phase invocations or dynamic phases. For reference, the total number of methods-level phases as well as the total number of dynamic method invocations are shown.

6.5.2 Variability within and between phases

The purpose of this section is to evaluate the variability within and between the phases. Our first metric is the Coefficient of Variation (CoV) which quantifies the variability within a phase, see Section 6.4. Figure 6.7 shows the CoV for the various benchmarks over various characteristics (CPI, L1 D-cache misses, L1 I-cache misses, and branch mispredictions). We observe that for the CPI and the branch mispredictions, the CoV is quite small. This indicates that the behaviour within a phase is quite stable for these characteristics. The I-cache behaviour on the other hand, is not very stable within the phases for *mpegaudio*. This can be due to the low I-cache miss rate for *mpegaudio* for which a similarly small variation exists. Indeed, a small variation in absolute terms, e.g.,

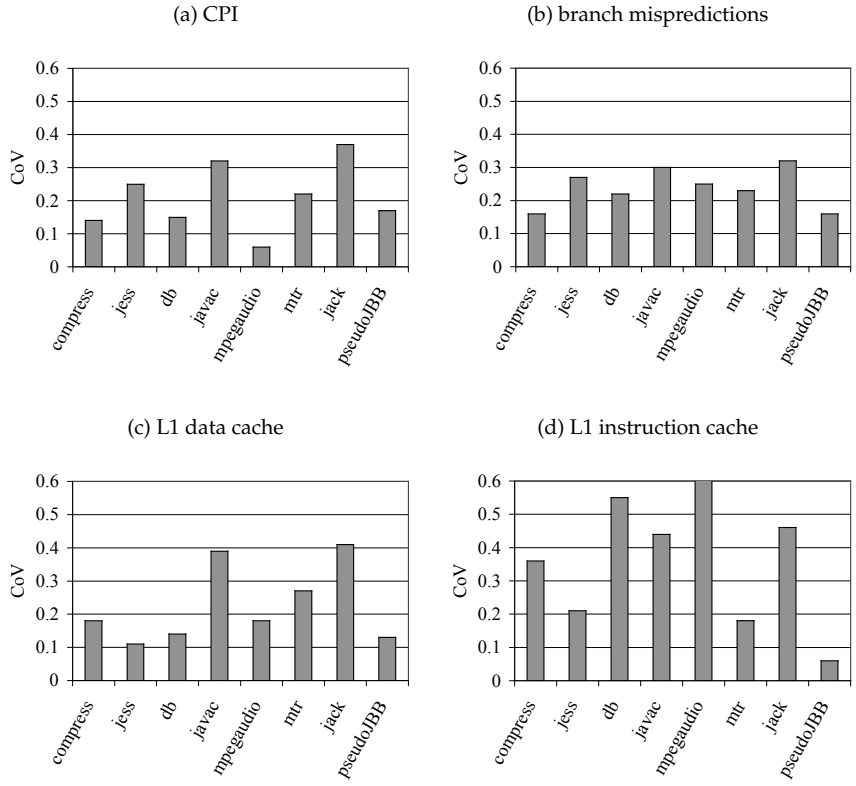
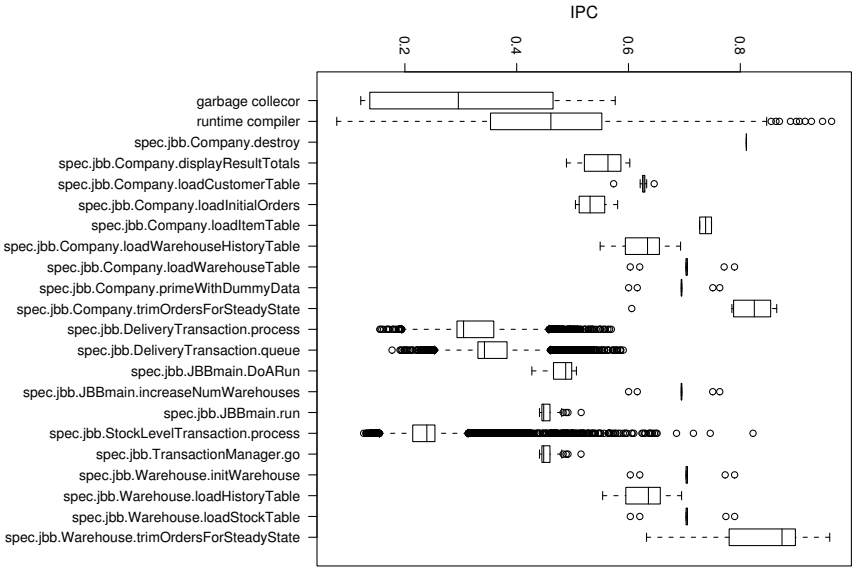


Figure 6.7: Accumulated weighted CoV values for the various benchmarks for four characteristics: (a) CPI, (b) branch mispredictions (c) L1 D-cache misses, and (d) L1 I-cache misses.

0.0001 versus 0.0002, can result in large variations in relative terms (100% in this example). As such, we do not consider this a major problem since analysts do not care about code having very low cache miss rates. Furthermore, if unstable behaviour for a given characteristic is undesirable, θ_{weight} and θ_{grain} can be adjusted so that more phases are selected and less variability will be observed within the phases.

To get a better view on the performance characteristics within and between phases, we use boxplots. Figures 6.8 and 6.9 show the performance characteristics for the various phases for *PseudoJBB*. On the vertical axis of these graphs, we display all the phases. The horizontal axes represent some performance metrics that were measured: IPC, L1 I-cache miss rate, L1 D-cache miss rate and the branch misprediction rate. For each phase, we display the mean value as the middle of the rectangular box. The borders of each box represent the standard deviation and the individual points above and under these boxes

(a) IPC



(b) Branch mispredictions

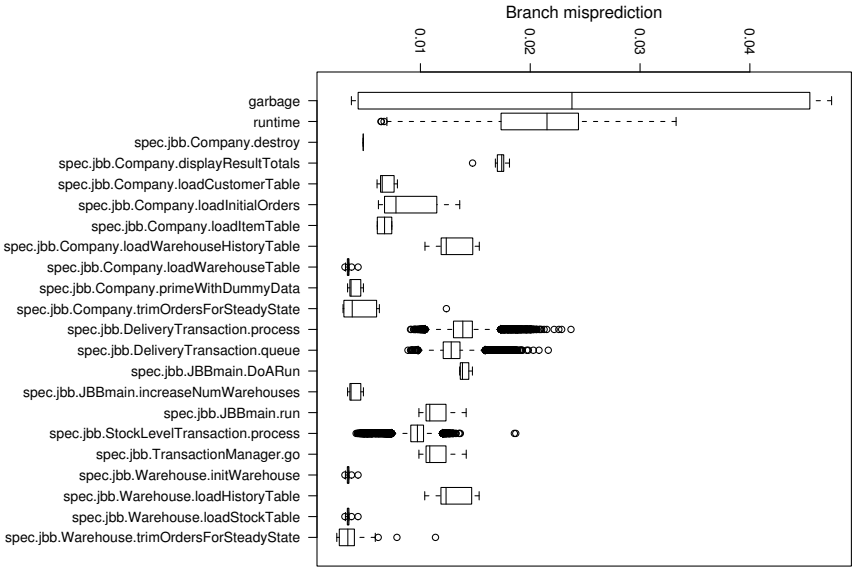
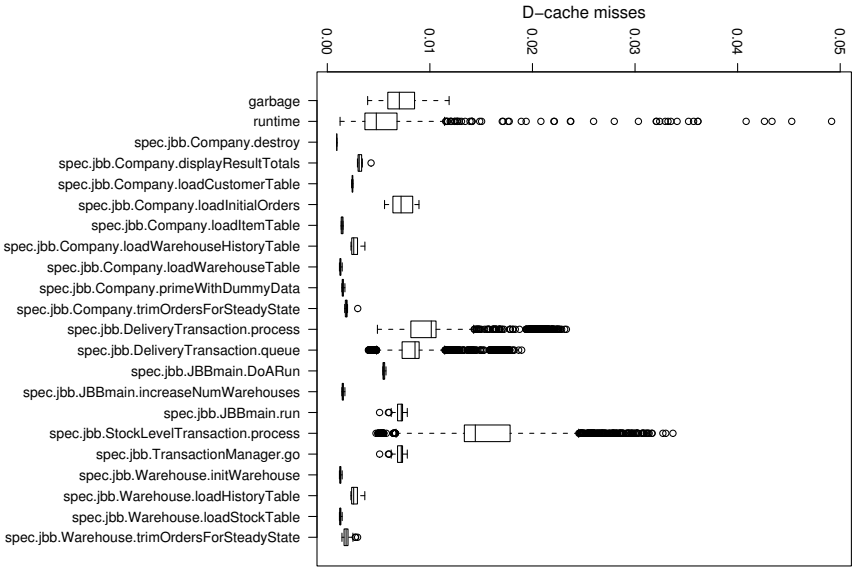


Figure 6.8: Boxplots showing the distribution for the phases of *PseudoJBB* on the following characteristics: (a) IPC, (b) mispredicted branches.

(a) L1 D-cache misses



(b) L1 I-cache misses

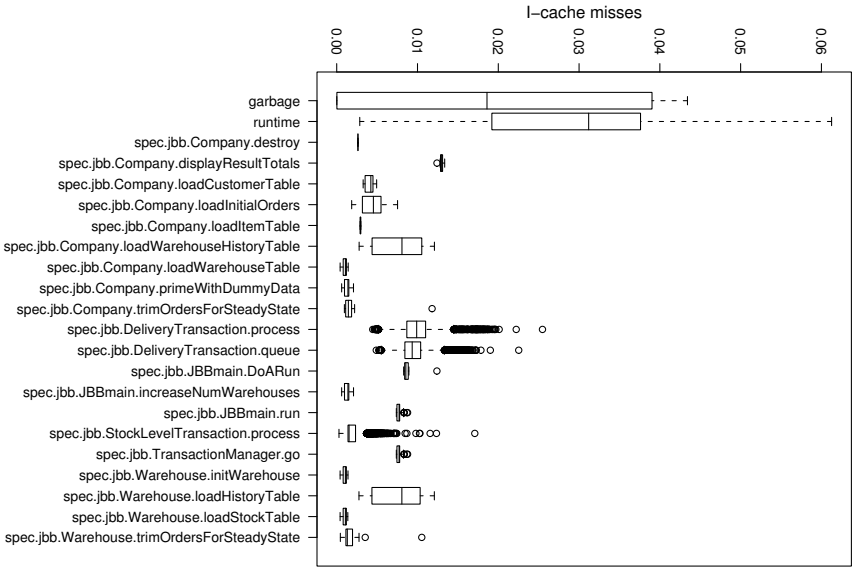


Figure 6.9: Boxplots showing the distribution for the phases of *PseudoJBB* on the following characteristics: (a) L1 D-cache misses, and (b) L1 I-cache misses.

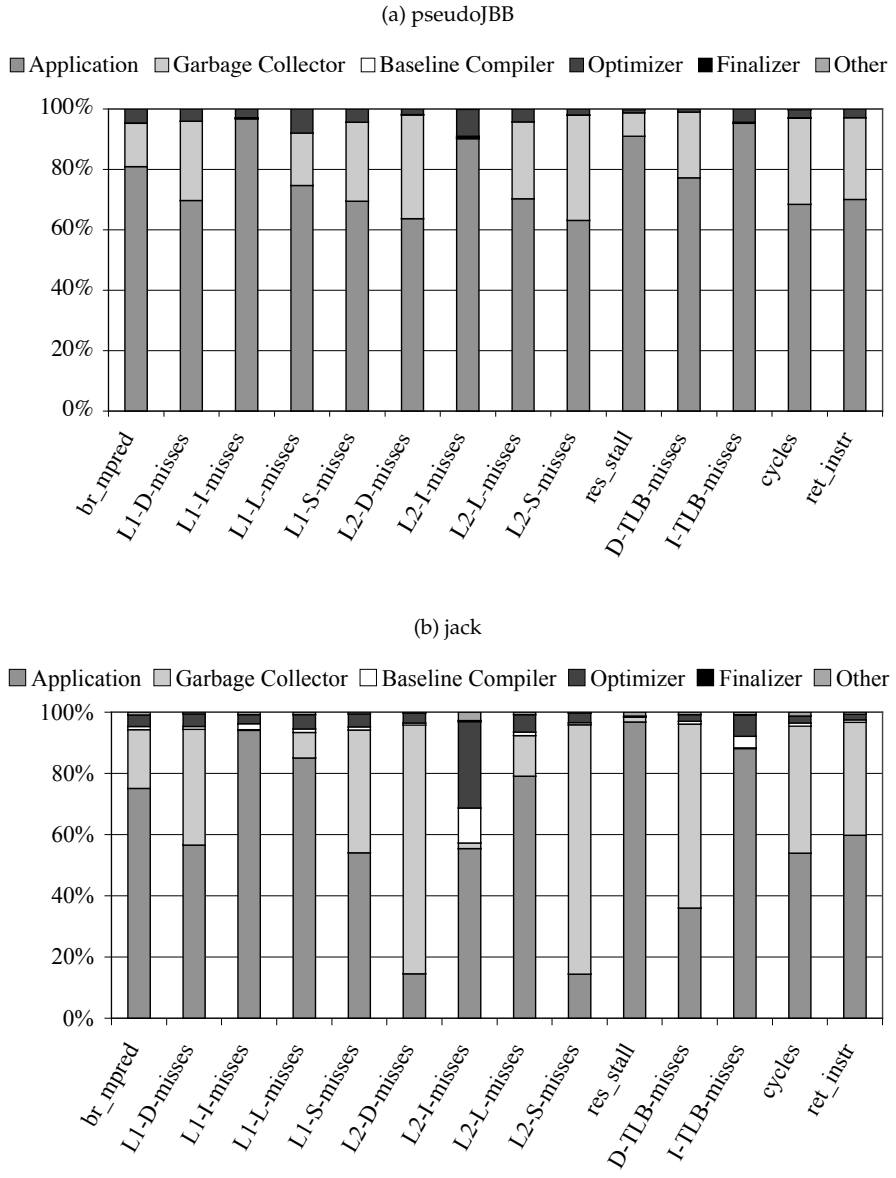


Figure 6.10: Breakdown of the performance characteristics for the application versus the VM components for *PseudoJBB* (a) and *jack* (b). Note that these graphs only show the results for a single heap size and a single garbage collector.

Benchmark	Configuration		ANOVA results		
	θ_{grain} (%)	θ_{weight} (%)	F-value	df	p-value
compress	8×10^{-6}	6×10^{-6}	37.94	(48, 2640)	$< 10^{-16}$
jess	1.0	1.0	1.74	(10, 664)	0.067
db	8×10^{-6}	6×10^{-6}	43.03	(49, 32339)	$< 10^{-16}$
javac	2×10^{-2}	6×10^{-3}	117.34	(20, 10749)	$< 10^{-16}$
mpegaudio	2×10^{-2}	2×10^{-3}	495.95	(26, 102894)	$< 10^{-16}$
mtrt	10^{-2}	10^{-3}	39.23	(26, 89004)	$< 10^{-16}$
jack	1.0	10^{-2}	63.56	(14, 2934)	$< 10^{-16}$
PseudoJBB	2×10^{-1}	2×10^{-4}	288.24	(19, 32206)	$< 10^{-16}$

Table 6.3: Results for ANOVA comparing the means for the observed characteristics. df denotes the degrees of freedom.

Benchmark	Program	GC	Finalizer	Compiler		Other
				Base	Opt	
compress	89.136	9.377	0.006	0.205	0.696	0.580
jess	56.835	39.641	0.003	0.919	1.914	0.688
db	92.211	6.991	0.002	0.128	0.455	0.213
javac	65.463	28.987	0.008	0.940	3.618	0.984
mpegaudio	85.000	7.999	0.002	0.559	5.821	0.620
mtrt	65.802	28.039	0.005	0.485	4.687	0.982
jack	53.905	41.556	0.015	0.941	2.317	1.265
PseudoJBB	73.348	22.974	< 0.001	0.091	3.532	0.063

Table 6.4: Percentage of the time spent in the application and the different VM components.

represent outliers, i.e., not within one standard deviation from the mean. This figure clearly shows that the performance characteristics can be quite different between phases. The variability within each phase on the other hand, is usually small. Notable exceptions to this are the garbage collector and the virtual machine compiler. The behaviour during a garbage collection is highly dependent on the heap organisation, the links between reachable objects, their sizes, etc. The behaviour during compilation is dependent on the bytecode, i.e., the method to compile, and the optimisations to use. This explains the larger variability within these phases.

We now quantify the variability between the phases versus the variability within the phases in a more rigorous way. This is done using an ANOVA test, see Section 6.4. In Table 6.3, we show the *maximum* p-value per benchmark over all characteristics. Recall that the lower p, the better. In our results, p

is smaller than 10^{-16} for nearly all benchmarks, from which we conclude that the mean values for each characteristic for the various phases are different at a significance level that almost reaches certainty. This means that the variability between the phases is significantly larger than the variability within the phases which proves that our off-line technique reliably identifies phases with dissimilar inter-phase behaviour and similar intra-phase behaviour. For *jess*, however, the p value reported in Table 6.3 is larger than for the other benchmarks. The higher p-values are due to the I-TLB and D-TLB miss rates, which do not show that much variability between the phases.

6.5.3 Analysis of method-level phase behaviour

A programmer analysing application behaviour will typically start from a high-level view of the program. Two of the first things one wants to analyse are where the time is spent, and whether potential bottlenecks are due to the application or the VM. In the first subsection, we look at the high-level behaviour of Java applications and compare it with the behaviour of the VM (GC, (JIT) compiler, etc.). Once the high-level behaviour has been understood, the next logical step is to investigate parts of the application into more detail. The subsequent subsection shows how the programmer can use the information collected by our framework to gain insight about the low-level behaviour of his program, and how our data can help identify and explain performance bottlenecks.

VM versus application behaviour

Figure 6.10(a) shows the number of events occurring in the application versus the VM. This is done here for *PseudoJBB*. We observe that most of the events occur in the application and not in the VM. Indeed, the total program execution spends 73% of its total execution time (expressed in cycles) in the application; the remaining 27% is spent in the VM. The time spent in the VM is partitioned in the time spent in the various VM components: compiler, optimiser, garbage collector, finaliser, and others. We observe that the most dominant part of the VM routines is due to the optimiser (3.5%) and the garbage collector (23%). This graph reveals several interesting observations. For example, although the optimiser is responsible for only 3.5% of the total execution time, it is responsible for a significantly larger proportion of the L1 D-cache load misses (6.3%) and L2 I-cache misses (13.7%). The garbage collector on the other hand, accounts for significantly more L2 D-cache misses (28%) than it accounts for execution time (21%). Another interesting result is that the garbage collector accounts for a negligible fraction of the L1 and L2 I-cache and I-TLB misses. This is due to the fact that the garbage collector has a small instruction footprint while accessing a large data set.

Figure 6.10(b) presents a similar graph for *jack*. The percentage of the total execution time spent in the application is 54%. Of the 46% spent in the VM,

41.5% is spent in the garbage collector, 2.3% in the optimiser, 0.9% in the baseline compiler and 1.3% in other VM routines, such as the thread scheduler, class loader, etc. These results confirm the specific behaviour of the garbage collector previously observed for *PseudoJBB*: low L1 and L2 I-cache and I-TLB miss rates and high L2 D-cache and D-TLB miss rates (due to writes). The baseline compiler and the optimiser show high L2 I-cache miss rates.

Table 6.4 presents the time spent in the application versus the time spent in the VM components for the SPECjvm98 and *PseudoJBB* benchmarks. The time spent in the application varies between 54% and 92% of the total execution time; the time spent in the garbage collector varies between 7% and 42% and the time spent in the optimiser varies between 0.4% and 5.8%. The execution time in the other VM components is negligible. We conclude that Java workloads spend a significant fraction of their total execution time in the VM, up to 46% for the long-running applications included in our study. The reason is the garbage collector used, i.e., CopyMS. With other algorithms, e.g., GenMS, less execution time is spent in the collector. Also, varying the heap size should be done to get a better idea of the time spent in the collector. Still, it is interesting to note that the three benchmarks (*compress*, *db*, and *mpegaudio*) for which the total execution time spent in the application is significantly larger than the average case (89%, 92% and 85%, respectively), were denoted as 'simple' benchmarks by Shuf et al. [99].

Application bottleneck analysis

Profilers provide a means for programmers to perceive the way their programs are performing. Our technique provides an easy way to the programmer to gain insight about the performance of their application at the micro-architectural level. That is, hardware performance counters can be linked to the methods in the source code. The results shown in this section help locate potential bottlenecks in the application, in particular those methods with a CPI value that is above the benchmark average.

Tables 6.5 to 6.8 present the major bottleneck phases for both the SPECjvm98 benchmarks and for *PseudoJBB*; the table depicts a subset of all phases only: we show the phases of which the total execution time takes more than 1% or 2% of the program execution time and of which the CPI is above the average CPI, or which otherwise display bad behaviour for a particular characteristic. This table shows the percentage of the total execution time that is spent in each phase, the average CPI in each phase, the cache miss rates and the branch misprediction rate. This information can be helpful in identifying why these phases suffer from such a high CPI. For example, high D-cache miss rates suggest that the programmer should try to improve the data memory behaviour for the given phases. We can make the following interesting observations – these are just a few examples to clarify the usefulness of linking microprocessor level information to source-code level methods.

- The `Compressor.compress` method in *compress* suffers from high D-cache miss rates. Optimisation of the data memory behaviour may be achieved, for example, by applying prefetching.
- From all the benchmarks, *mtrt* has a method with the most mispredicted branches: `Scene.RenderScene`. This method contains two nested loops, iterating over all pixels in the scene to be rendered. Inside the loop there are a number of conditional branches and a call to `Scene.Shade`. In turn, the latter shows bad branch behaviour due to numerous (nested) tests that are conducted to decide on the colour of the pixel that is being rendered. This behaviour may be optimised by changing the code layout to improve the branch predictability. Hence, this might be a typical VM improvement.
- Poor I-cache behaviour can be observed for e.g. the `expansion_choices` method in *jack*.
- For the SPECjvm98 benchmarks, the GC shows a very consistent behaviour, with a CPI that remains around 1.77. Also, GC shows a very good I-cache behaviour both on L1 and L2. This due to the fact that (i) GC usually can take quite some time, hence the initial cache misses can be made up for by a longer execution time, and (ii) GC code is usually quite compact.

Phase	Time (%)	CPI	L1-D	L1-I	L2-D	L2-I	br_mpred
compress							
garbage collector	9.3773	1.7778	4.04	0.02	2.59	0.01	4.39
Compressor.compress	58.3916	1.7447	22.91	0.01	4.36	< 0.01	7.28
Decompressor.decompress	25.2042	0.9242	2.53	< 0.01	0.12	< 0.01	4.83
benchmark average	n/a	1.4830	12.25	0.10	2.01	0.04	5.69
jess							
Jesp.parse	1.1021	1.8701	4.52	5.91	1.04	1.71	14.55
garbage collector	39.6413	1.7647	4.02	0.02	2.58	0.01	4.33
Rete.Run	53.8732	1.1796	4.92	0.51	0.45	0.05	4.51
benchmark average	n/a	1.3959	4.66	0.68	1.12	0.17	4.73
db							
Database.shell_sort	85.5593	5.1134	26.42	0.02	18.01	0.01	4.87
Database.remove	4.5821	2.7155	11.33	0.10	6.28	0.06	1.36
garbage collector	6.9912	1.7989	3.92	0.03	2.45	0.01	4.23
Database.set_index	2.3873	1.5749	5.74	0.08	3.38	0.04	0.08
benchmark average	n/a	3.9847	4.71	0.05	3.13	0.01	1.07

Table 6.5: Interesting methods from the SPECjvm98 *compress*, *jess*, and *db* benchmarks. The L1 and L2 I-cache miss rates, L1 and L2 D-cache miss rates and the branch misprediction rate are given as the number of events per 1,000 instructions.

Phase	Time (%)	CPI	L1-D	L1-I	L2-D	L2-I	br_mpred
javac							
SourceClass.check	7.0644	2.2501	8.06	13.13	1.74	1.75	23.2
SwitchStatement.check	1.4973	1.9129	7.91	13.21	0.76	0.49	22.87
garbage collector	28.9874	1.8059	4.48	0.02	2.55	0.01	4.76
SourceClass.compileClass	26.0361	1.7408	5.14	6.53	0.98	0.85	16.26
Assembler.collect	1.8285	1.6994	4.96	4.58	1.23	0.52	13.64
Parser.parseClass	22.6849	1.4789	2.93	5.52	0.54	0.44	18.26
ConstantPool.write	5.3863	1.0231	1.20	0.19	0.33	0.08	6.26
benchmark average	n/a	1.6747	4.28	4.48	1.32	0.66	13.26

Table 6.6: Interesting methods from the SPECjvm98 *javac* benchmark. The L1 and L2 I-cache miss rates, L1 and L2 D-cache miss rates and the branch misprediction rate are given as the number of events per 1,000 instructions.

Phase	Time (%)	CPI	L1-D	L1-I	L2-D	L2-I	br_mpred
mpegaudio							
garbage collector	7.9994	1.7795	4.15	0.03	2.62	0.01	4.71
lb.read	20.7281	0.8602	1.41	1.17	0.01	< 0.01	6.38
t.O	75.1119	0.8430	0.82	0.49	< 0.01	< 0.01	2.29
benchmark average	n/a	0.8157	1.07	0.47	0.03	0.02	3.25
mtrt							
Scene.RenderScene	1.9640	2.3249	12.32	18.74	0.21	0.25	35.98
garbage collector	28.0391	1.7829	3.73	0.02	2.40	< 0.01	4.47
Scene.GetLightColor	23.7782	1.4919	9.74	3.29	0.68	0.03	8.95
Scene.Shade	36.2558	1.3496	7.21	2.84	0.42	0.05	11.42
Scene.ReadPoly	2.4275	1.2909	1.52	3.32	0.12	0.10	8.88
benchmark average	n/a	1.5389	8.27	2.66	1.03	0.07	7.18

Table 6.7: Interesting methods from the SPECjvm98 *mpegaudio* and *mtrt* benchmarks. The L1 and L2 I-cache miss rates, L1 and L2 D-cache miss rates and the branch misprediction rate are given as the number of events per 1,000 instructions.

Phase	Time	CPI	L1-D	L1-I	L2-D	L2-I	br_mpred
jack							
Jack_the_Parser_Generator._Jack3_1	2.34092%	1.9655	5.84	5.19	0.57	0.22	11.53
garbage collector	41.5561%	1.7741	4.04	0.02	2.59	0.01	4.36
Jack_the_Parser_Generator.production	1.87112%	1.7039	4.38	7.41	0.51	0.73	15.16
Jack_the_Parser_Generator.jack_input	2.8412%	1.6014	3.59	6.26	0.38	0.56	13.69
Jack_the_Parser_Generator.expansion_choices	20.5098%	1.5546	4.46	7.54	0.22	0.23	15.94
Jack_the_Parser_Generator.java_declarations_and_code	19.4081%	1.3648	2.70	5.1	0.11	0.09	12.64
Jack_the_Parser_Generator_Internals.db_process	2.78693%	1.2737	2.61	1.61	0.59	0.28	4.74
ParseGen.build	2.6689%	1.1157	2.27	0.37	0.69	0.06	2.41
benchmark average	n/a	1.5976	3.83	3.58	1.19	0.24	9.58
PseudoJBB							
DeliveryTransaction.process	2.7597%	3.0722	8.74	9.95	6.45	2.61	17.32
garbage collector	22.9744%	2.1581	5.76	0.03	3.59	< 0.01	4.35
TransactionManager.go	57.9074%	2.1219	6.77	7.77	2.91	0.75	11.08
benchmark average	n/a	2.046	6.02	5.02	2.69	0.57	9.13

Table 6.8: Interesting methods from the SPECjvm98 *jack* and SPECjbb2000 (as observed in *PseudoJBB*) benchmarks. The L1 and L2 I-cache miss rates, L1 and L2 D-cache miss rates and the branch misprediction rate are given as the number of events per 1,000 instructions.

6.6 Related work

In this section we briefly discuss related work. We make a distinction between work that was done on characterising Java workloads, and work that is aimed at phase classification and detection techniques.

6.6.1 Java workload characterisation

Cain et al. [31] characterise the Transaction Processing Council's TPC-W web benchmark which is implemented in Java. TPC-W is designed to exercise the web server and transaction processing system of a typical e-commerce web site. They used both hardware execution (on an IBM RS/6000 S80 server with 8 RS64-III processors) and simulation in their analysis.

Karlsson et al. [64] study the memory system behaviour of Java-based middleware. To this end, they study both the SPECjbb2000 and SPECjAppServer2001 benchmarks on real hardware as well as through simulation. For the real hardware measurements, they use the hardware counters on a 16-processor Sun Enterprise 6000 multiprocessor server. They measure performance characteristics over the complete benchmark run and make no distinction between the VM and the execution phases of the application.

Luo et al. [72] compare SPECjbb2000 versus SPECweb99, VolanoMark and SPEC CPU2000 on three different hardware platforms: the IBM RS64-III, the IBM POWER3-II and the Intel Pentium III. All measurements were done using performance counters and measure aggregate behaviour.

Dufour et al. [42] present a set of architecture-independent metrics for describing dynamic characteristics of Java applications. All these metrics are bytecode-level program characteristics and measure program size, the intensiveness of various data structures (arrays, pointers, floating-point operations), memory use, concurrency, synchronisation and the degree of polymorphism.

Dmitriev [40] presents a bytecode-level profiling tool for Java applications, called JFluid. During a typical JFluid session, the VM is started with the Java application without any special preparation. Subsequently, the tool is attached to the VM, the application is instrumented, the results are collected and analysed on-line, and the tool is detached from the VM. The instrumentation is done by injecting instrumentation bytecodes into methods of a running program. In JFluid, the user needs to specify which call subgraph, called a *task* by Dmitriev, from an arbitrary root method is to be instrumented. This method has two major differences with our approach: (i) we do not operate at the bytecode level but at the lower microprocessor level, and (ii) we provide a means to automatically detect these *tasks*. This relieves the user from manually selecting major tasks of execution.

Maebe et al. [75] present Javana, a framework for building customised vertical profiling tools. This allows a programmer to correlate performance

events – for example, cache misses – with source code lines using a binary instrumentation tool, DIOTA [74].

Sweeney et al. [108] present a system to measure microprocessor level behaviour of Java workloads. To this end, they generate traces of hardware performance counter values while executing Java applications. This is done for each Java thread and for each microprocessor on which the thread is running. The latter can be useful in case of a multiprocessor environment. The infrastructure for reading performance counter values used by Sweeney et al. is exactly the same as the one used in this chapter – using HPM in the Jikes RVM – except for the fact that our measurements are done on an IA-32 ISA platform opposed to the PowerPC ISA platform. Sweeney et al. read the performance counter values on every virtual context switch in the VM. This information is collected for each virtual processor and for each Java thread, and written in a per virtual processor record buffer. Sweeney et al. also present a tool for graphically exploring the performance counter traces. The major difference between the work by Sweeney et al. and our work, is that we collect performance counter values on a per-phase basis as opposed to the timing-driven approach of taking one sample on every virtual context switch. The benefit of measuring performance counter values on a per-phase basis is that performance counter values can be easily linked to the code that is executed in the phase. We believe this is particularly useful for analysis in general, and for application and VM developers in particular. Moreover, our approach is more general than the approach by Sweeney et al. since the information we obtain can be easily transformed to behavioural information over time. This can be done by ordering our information on a time-line basis. The advantage Sweeney et al. offer compared to our approach is that theirs is an online tool, which does not require two runs to characterise program execution.

In [54], Hauswirth et al. extended the work of [108]. They present a vertical profiling approach to examine performance across the complete execution stack when a Java application is executing. For this, they use hardware performance monitors that are available in Jikes RVM, and they add software monitors to observe interesting events, such as garbage collection, compilation, synchronisation and OS-interaction. They correlate performance hits with the underlying events. In [55], they take the technique one step further and automate the alignment of traces obtained from each of the vertical layers. They also automate the finding of correlation between events and performance hits, rather than having to sort this out manually.

6.6.2 Program phases

Several techniques that have been proposed in the recent literature to detect program phases that divide the total program execution in fixed intervals. For each interval, program characteristics are measured during program execution. When the difference in program characteristics between two consecutive intervals exceeds a given threshold, the algorithm detects a phase change. These approaches are often referred to as *temporal techniques*. The proposed temporal techniques all differ in what program characteristics need to be measured over the fixed interval. Balasubramonian *et al.* [9] compute the number of dynamic conditional branches executed. A phase change is detected when the difference in branch counts between consecutive intervals exceeds a threshold. This threshold is adjusted dynamically during program execution to match the program's execution behaviour. Dhodapkar and Smith [38] use the instruction working set or the instructions executed at least once. Since representing a complete working set is impractical, especially in hardware, the authors propose working set signatures which are lossy-compressed representations of working sets. Working set signatures are compared using a relative signature distance. A program phase change is detected when the relative signature distance between consecutive intervals exceeds a given threshold. Sherwood et al. [97, 98] use basic block vectors (BBVs) to identify phases. A BBV is a vector in which the elements count the number of times each static basic block is executed in the fixed interval. These BBVs are weighted by the number of instructions in the given basic block. A phase change is detected when the Manhattan distance between two consecutive intervals exceeds a given threshold. They consider both static and dynamic methods for identifying phases in [97] and [98], respectively. The purpose of their static phase classification approach was to identify equally behaving intervals throughout a program execution so that one single representative interval for each phase can be used for efficient simulation studies. In a follow-up study, Lau et al. [66] study several structures for classifying program execution phases. They study approaches using basic blocks, loops, procedures, opcodes, register usage and memory address information. In contrast to the previously mentioned approaches which all use micro-architecture-independent characteristics – i.e. the metrics are only dependent on the instruction set architecture (ISA) and not on the micro-architecture – Duesterwald et al. [41] use micro-architecture-dependent characteristics to detect phases. The metrics used by them are the instruction mix, the branch prediction accuracy, the cache miss rate and the number of instructions executed per cycle (IPC). These metrics are measured using performance counters over fixed intervals of 10 milliseconds.

Next to temporal phase detection approaches, there exist a number of approaches that do not use fixed intervals. Balasubramonian et al. [9] consider procedures to identify phases. They consider non-nested and nested procedures as phases. A non-nested procedure is a procedure that includes its complete call graph, i.e., including all the methods it calls, as is done in this

chapter. A nested procedure does not include its callees. They concluded that non-nested procedures are better performing than nested procedures. Huang et al. [60] also use procedures to identify phases. The method used in our work to identify method-level phases of execution – using θ_{weight} and θ_{grain} – is based on the approach proposed by Huang et al. Next to this static approach, they also propose a hardware-based call stack mechanism to identify program phase changes. Our work differs from the one by Huang et al. for at least three reasons. First, we explore the technique for detecting phases in more detail by quantifying the overhead and coverage as a function of θ_{weight} and θ_{grain} . Huang et al. chose fixed θ_{weight} = 5% and θ_{grain} = 1,000 cycles in their experiments. Second, we study Java workloads whereas Huang et al. studied SPEC CPU2000 benchmarks. Java workloads provide several additional challenges over C-style workloads because of the managed run-time environment. Third, the focus of the work by Huang et al. was on exploiting phase behaviour for energy-efficient computing. The focus of our work is on using phase behaviour to increase the understanding of a program's time varying behaviour.

6.7 Conclusions

Java applications often have a quite complex interaction with the virtual machine executing them. If a programmer wants to gain insight into the performance of his application, and improve its performance, it does not suffice to observe global performance. In fact, because global performance numbers do not allow a distinction between application and virtual machine components, one cannot simply compare two performance numbers and claim that the changes have actually improved the application itself. Consequently, automatic tools to characterise these software stacks become paramount during performance analysis.

In this study we considered method-level phases in Java applications. The goal was to identify methods such that similar behaviour is observed within each phase, and dissimilar behaviour is observed between the phases. We presented a three-step analysis framework. In the first step, we measure the execution time of all non-trivial methods using the hardware performance monitors available on modern hardware. In the second step, we analyse this information off-line and determine a set of methods that are considered phases in the program. In the last step, we characterise these phases with respect to a number of performance metrics, such as IPC, cache miss rate, branch misprediction rate, etc.

With this framework, we investigated the phase behaviour of both the SPECjvm98 and SPECjbb2000 benchmark suite.

The key contributions we present in this chapter are the following. First, we have shown that phases exhibiting uniform behaviour in Java application can be found at the method level. Second, we provided a semi-automatic way

for detecting these phases. Third, we have shown that our technique can help a programmer detect bottlenecks by providing more in-depth information regarding the performance of an application compared to global performance numbers and guide him in optimising his application.

Chapter 7

Conclusion

Witness! – **Karsa Orlong**

7.1 Summary

In this dissertation we touched upon three pitfalls researchers and benchmarkers need to take into account when analysing the performance of various aspects comprising a Java workload.

We have illustrated in Chapter 2 that a Java workload exhibits complex interactions between the virtual machine and the application, a fact that is influenced by the input set to the application. Clearly, being unaware of this interaction can lead a researcher toward generalising conclusions drawn from experiments that either consider but one or two virtual machines or use a small input set. In the former case the results will not be representative for other virtual machine environments, in the latter case one makes the mistake of assuming small input sets are representative for larger input sets.

In Chapters 3 through 5, we uncovered the pitfall associated with prevalent data analysis techniques. Our claim is that there is no substitute for rigorous statistical analysis. The reason is that Java workloads exhibit nondeterminism as a consequence of using a virtual machine to execute the application which needs JIT compilation, thread scheduling, garbage collection, optimisation, etc. We formulated a strategy to deal with both start-up and steady-state performance. We showed that prevalent approaches can lead to either misleading or even incorrect conclusions. Of course, it would be too bold to claim that results that were published in the past are in fact wrong. However, one needs to take care interpreting the results and be aware of the possibility that in some cases, reported results may not reflect what one

may observe in practice. We also showed that replay compilation does not always agree with non-controlled compilation. In some cases, admittedly, replay compilation might be beneficial to isolate parts of the virtual machine – i.e., simplify the complex interactions between virtual machine components – but using a single compilation plan, either optimal or majority, is no match for using a rigorous statistical analysis using multiple compilation plans. We advocate the use of multiple compilation plans with a matched-pair comparison to obtain tight confidence intervals, as opposed to using a single plan.

Finally, we have showed that it is more useful to zoom into the application, for example by exploiting the method-level phase behaviour to gain insight in the time-varying behaviour of a Java application. Once again, the complex nature of a Java workload compels us to separate application performance from the performance of virtual machine components, which can be achieved by applying our technique.

The realisation that there are pitfalls lurking around the bend, hopefully gives researchers the incentive to set up experiments in a careful manner. We would at least like to see the following in research practiced henceforth. First of all, it is important to carefully describe the experimental setup, leaving out no details. We do not contend the fact that computer science experiments may be among the hardest to reproduce, but reproducibility is a characteristic of a good scientific methodology. Second, there is to be no excuse for benchmark subsetting, other than the inability to execute a benchmark on a given platform. Finally, we encourage researchers to use a sound statistical data analysis – even for experiments where little or no variance is expected.

7.2 Future work

Research is never quite finished. We have trod but the beginning of a path toward better understanding Java performance and its associated issues and pitfalls. One particular challenging issue that remains is the quantification of steady-state behaviour. Deciding at which point an application reaches steady-state is yet an unsolved problem. There is no telling if at any time in the future other methods will be proposed and accepted for optimisation. In a similar vein, benchmarking very long running applications that execute for days, weeks, or even months, is hard to do in the rigorous way we propose. For these applications, doing multiple runs is often impractical, if not impossible.

Virtualisation does not end with the Java virtual machine. Nor does it stop at managed runtime environments for programming languages. A new challenge has made its way back to the front of computer science research: virtualisation at the system level. In today's service-oriented computerised world, security (sandboxing, system isolation, etc.), cost-reduction, efficiency, and server consolidation are becoming increasingly important. Consequently, old ways to share the power of a microprocessor – bearing multiple cores

nowadays – are becoming increasingly popular. The best known examples of system virtual machines are probably VMWare, which is an application-level system virtual machine, and Xen, which is a paravirtualised VM. Modern microprocessors offer support for system virtualisation, such as the IBM Power-5 and the SUN Niagara-1. The latest products from both AMD and Intel also offer virtualisation extensions that ease the use of system VMs and reduce the overhead when using such environments.

The crux of the matter is that we are dealing with an increasingly complex system. It is easy to see that the execution stack for the application on top of the stack, which is the only thing of real interest to the end-user, grows ever larger. If we consider the entire execution stack of the physical machine, there can be multiple processors – each with multiple cores – shielded from the guest OSes by a hypervisor. Alternatively, multiple system VMs, each with their guest OS, run on top of a host OS. In both cases, user applications run on the guest OS. It is hard to evaluate performance on such a system. Consequently, it is hard to manage resources due to the numerous interactions between all layers (application, guest OS, host OS, hypervisor) of a virtualised environment.

Appendix A

Setup

In this chapter we describe the benchmarks, virtual machines and platforms we used in this dissertation.

A.1 Benchmarks

We have used three benchmarks suites: SPECjvm98 [103], SPECjbb2000 [102], and DaCapo [17]. In this section we give a description of each of these suites.

SPECjvm98 is a client-side Java benchmark suite consisting of seven benchmarks, listed in Table A.1. For each of these, SPECjvm98 provides three inputs: `s1`, `s10` and `s100`. Contradictory to what the input set names suggest, the size of the input set does not increase exponentially. For some benchmarks, a larger input increases the problem size. For other benchmarks, a larger input executes a smaller input multiple times. SPECjvm98 was designed to evaluate combined hardware (CPU, caches, memory, etc.) and software aspects (virtual machine, kernel activity, etc.) of a Java environment. However, the benchmarks do not include graphics, networking or AWT (window management). All benchmarks can be executed on a virtual machine conforming to the Java 1.1 API.

SPECjbb2000 is a server-side benchmark representing a three-tier transaction system, where the user interaction is simulated by random input selection and the third tier, the database, is represented by a set of binary trees. The benchmark focuses on the business logic found in the middle tier. It is loosely based on the IBM pBOB benchmark [11]. The workload of the benchmark can be determined by varying the number of warehouses or by changing the amount of time (seconds) the benchmark is allowed to run.

In this dissertation we use a modified version of SPECjbb2000, also known as PseudoJBB. The main difference with SPECjbb2000 is that the run time is not determined in seconds but in the number of transactions that are con-

Benchmark	Information
compress	A compression program, using a LZW method ported from 129.compress in the SPECCPU95 suite. Unlike 129.compress, it processes real data from several files. The various inputs are obtained by performing a different number of iterations through various input files.
jess	An expert shell system, adapted from the CLIPS system. The various inputs consist of a set of puzzles to be solved, with varying degrees of difficulty.
db	The benchmark performs a set of database requests on a memory resident database of 1MiB. The various inputs are obtained by varying the number of requests to the database.
javac	This is the JDK 1.0.2 source code compiler. The various inputs are obtained by making multiple copies of the same input files.
mpegaudio	A commercial application decompressing MPEG Layer-3 audio files. The input consists of about 4MiB of audio data.
mtrt	A raytracer using two threads to render a scene. The various inputs are determined by the problem size.
jack	An early version of JavaCC which is a Java parser generator. The various inputs make several passes through the same data.

Table A.1: The SPECjvm98 benchmarks we use in this dissertation.

ducted for each set of warehouses. This means that the execution is repeatable across different machines, i.e., the workload executed remains the same, independent of the speed of the machine.

DaCapo is a benchmark suite developed to include real-world applications that have non-trivial memory loads and that are representative for Java applications. At the moment of writing, the suite (version 2006-10-MR2) includes the benchmarks listed in Table A.2. Unfortunately not all these benchmarks could be executed on the primary virtual machine, namely the Jikes RVM from the SVN repository February 12th 2007. We could only run the following, and these were used throughout Chapters 2 to 5: *antlr*, *bloat*, *fop*, *hsqldb*, *jython*, *luindex*, *pmd*.

Benchmark	Information
antlr	parses one or more grammar files and generates a parser and lexical analyser for each.
bloat	performs a number of optimisations and analysis on Java bytecode files
chart	uses JFreeChart to plot a number of complex line graphs and renders them as PDF
eclipse	executes some of the (non-gui) jdt performance tests for the Eclipse IDE
fop	takes an XSL-FO file, parses it and formats it, generating a PDF file.
hsqldb	executes a JDBCbench-like in-memory benchmark, executing a number of transactions against a model of a banking application
jython	inteprets a the pybench Python benchmark
luindex	Uses lucene to indexes a set of documents; the works of Shakespeare and the King James Bible
lusearch	Uses lucene to do a text search of keywords over a corpus of data comprising the works of Shakespeare and the King James Bible
pmd	analyses a set of Java classes for a range of source code problems
xalan	transforms XML documents into HTML

Table A.2: The DaCapo benchmarks, from which we use *antlr*, *bloat*, *fop*, *hsqldb*, *jython*, *luindex*, and *pmd* in this dissertation.

A.2 Virtual machines

In our study, we have used five virtual machine configurations which are tabulated in Table A.3.

Both the SUN JRE 1.5 and the Blackdown JRE 1.4.1 virtual machines are based on the SUN Hatsheput virtual machine core [107]. HotSpot uses a mixed scheme of interpretation, Just-in-Time (JIT) compilation and optimisation to execute Java applications. The degree of optimisation can be specified by choosing either client mode or server mode. In client mode, the virtual machine performs fewer runtime optimisations resulting in a limited application boot time and a reduced memory footprint. In server mode, the virtual machine performs classic code optimisations as well as optimisations that are more specific to Java, such as null-check and range-check elimination. It is also interesting to note that HotSpot maps Java threads to native OS threads. The garbage collector uses a fully accurate, generational copying scheme. New objects are allocated in the 'nursery' and moved to the 'old object' space when

Virtual machine	Configuration
SUN JRE 1.5	Hotspot with the client JIT compiler (mixed-mode), generational stop-the-world GC with a nursery, a tenure space and a permanent space
Blackdown JRE 1.4.1	Hotspot with the client JIT compiler (mixed-mode), generational GC with a nursery and a tenure space
Jikes RVM	adaptive optimising compilation-only mode, generational mark-sweep GC (GenMS)
JRockit 1.4.1	Adaptive optimising mode, generational copying GC
IBM J9	Interpretation plus a JIT compiler (mixed-mode), non-generational mark-sweep compacting GC

Table A.3: Java virtual machines used to study the interaction between the VM and the Java application.

the 'nursery' is collected. Objects in the 'old object' space are reclaimed by a mark and sweep compacting strategy.

BEA Weblogic's JRockit [13] is a virtual machine that is targeted at server-side Java. JRockit compiles methods upon their first invocation. At runtime, statistics are gathered and hot methods are scheduled for optimisation. The optimised code replaces the old code while the virtual machine keeps running. This way, an adaptive optimisation scheme is realised. JRockit uses a mixed threading scheme, called Thin Thread, in which n Java threads are multiplexed on m native threads. The virtual machine comes with four possible garbage collection strategies. We have used the generational copying version in our experiments.

Jikes [3] is a Research Virtual Machine (RVM)—Previously known as Jalapeño—that is targeted at server-side Java applications. Jikes is written entirely in Java and uses compilation throughout the entire execution (no interpretation). It is possible to configure the JikesRVM in different compiling modes: a baseline compiler, an optimising compiler and an adaptive compiler. We have used the adaptive mode in our experiments, which amounts to baseline compiling the method upon first invocation and optimising when the method is deemed sufficiently hot by the VM analysis component. The threading system multiplexes n Java threads to m native threads, also referred to as virtual processors. There is a range of garbage collection strategies available for this virtual machine. Among them are state-of-the-art copying, mark-and-sweep and generational collectors as well as combinations of these strategies.

The IBM J9 [61] also uses a mixed strategy by employing IBM's JIT compiler as well as IBM's Mixed Mode Interpreter (MMI). This is IBM's current production VM.

A.3 Platforms

The main platform used in this dissertation is the AMD Athlon XP. In this section we describe the main features of this microprocessor that are of interest to this dissertation.

The AMD Athlon XP is a member of the AMD K7 family [1, 39]. Details on this architecture are given in Table A.4. The AMD K7 is a superscalar microprocessor implementing the IA-32 instruction set architecture (ISA). It has a pipelined microarchitecture in which up to three x86 instructions can be fetched. These instructions are fetched from a large predecoded 64KB L1 instruction cache (I-cache). For dealing with the branches in the instruction stream, branch prediction is done using a global history (gshare) based taken/not-taken branch predictor, a branch target buffer (BTB) and a return address stack (RAS). Once fetched, each (variable-length) x86 instruction is decoded into a number of simpler (and fixed-length) macro-ops. Up to three x86 instructions can be translated per cycle.

These macro-ops are then passed to the next stage in the pipeline, the instruction control unit (ICU) which basically consists of a 72-entry reorder buffer. From this reorder buffer, macro-ops are scheduled into an 18-entry integer scheduler and a 36-entry floating-point scheduler for integer and floating-point operations, respectively. The 18-entry integer scheduler is organised as a collection of three 6-entry deep reservation stations, each reservation station serving an integer execution unit and an address generation unit. The 36-entry floating-point scheduler (FPU: floating-point unit) serves three floating-point pipelines executing x87, MMX and 3DNow! operations. In the schedulers, the macro-ops are broken down to ops which can execute out-of-order. Next to these schedulers, the AMD K7 microarchitecture also has a 44-entry load-store unit. The load-store unit consists of two queues, a 12-entry queue for L1 D-cache load and store accesses and a 32-entry queue for L2 cache and memory load and store accesses—requests that missed in the L1 D-cache. The L1 D-cache is organised as an eight-bank cache having two 64-bit access ports.

Another interesting aspect of the AMD K7 microarchitecture is the fact that the L2 unified cache is an exclusive cache. This means that cache blocks that were previously held by the L1 caches but had to be evicted from L1, are held in L2. If the newer cache block that is to be stored in L1 previously resided in L2, that cache block will be evicted from L2 to make room for the L1 block, i.e., a swap operation is done between L1 and L2. If the newer cache block that is to be stored in L1 did not previously reside in L2, a cache block will need to be evicted from L2 to memory.

component	sub-component	description
memory hierarchy	L1 I-cache	64KB two-way set-associative, 64-byte lines, LRU replacement with next line prefetching
	L1 D-cache	64KB two-way set-associative, 8 banks with 8-byte lines, LRU write-allocate, write-back, two access ports 64 bits each
	L2 cache	64KB two-way set-associative, unified, on-chip, exclusive
	L1 I-TLB	24 entries, fully associative
	L2 I-TLB	256 entries, four-way set-associative
	L1 D-TLB	32 entries, fully associative
	L2 D-TLB	256 entries, four-way set-associative
branch prediction	BTB	branch target buffer, two-way set-associative, 2048 entries
	RAS	return address stack, 12 entries
	taken/not-taken	gshare 2048-entry branch predictor with 2-bit counters
system design	bus	200MHz, 1.6GiB per second
pipeline stages	integer	10 cycles
	floating-point	15 cycles
integer pipeline	pipeline 1	integer execution unit and address generation unit also allows integer multiply
	pipeline 2	integer execution unit and address generation unit
	pipeline 3	idem
floating-point pipeline	pipeline 1	3DNow! add, MMX ALU/shifter and floating-point add
	pipleine 2	3DNow!/MMX multiply/reciproce, MMX ALU and floating-point multiply/divide/square root
	pipeline 3	floating-point constant loads and stores

Table A.4: The AMD Athlon XP microprocessor summary.

Bibliography

[1] Advanced Micro Devices, Inc. *AMD Athlon Processor x86 Code Optimization Guide*, February 2002. `http://www.amd.com`.

[2] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., 1986.

[3] B. Alpern, C.R. Attanasio, J.J. Barton, M.G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S.J. Fink, D. Grove, M. Hind, S.F. Hummel, D. Lieber, V. Litvinov, M.F. Mergen, T. Ngo, J.R. Russell, V. Sarkar, M.J. Serrano, J.C. Shepherd, S.E. Smith, V.C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño Virtual Machine. *IBM Systems Journal*, 39(1), pages 211–238, February 2000.

[4] M. Arnold, S. Fink, D. Grove, M. Hind, and P.F. Sweeney. Adaptive optimization in the Jalapeño JVM. In *OOPSLA*, pages 47–65, Oct. 2000.

[5] M. Arnold, M. Hind, and B.G. Ryder. Online feedback-directed optimization of Java. In *OOPSLA*, pages 111–129, Nov. 2002.

[6] P.C. Austin and J.E. Hux A brief note on overlapping confidence intervals In *Journal of Vascular Surgery*, 36(1), pages 194–195, 2002

[7] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An infrastructure for computer system modeling. In *IEEE Computer*, 35(2):59–67, February 2002.

[8] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *PLDI 2000*, pages 1–12. ACM, 2000.

[9] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures. In *MICRO*, pages 245–257. ACM, 2000.

[10] K. Barabash, Y. Ossia, and E. Petrank. Mostly concurrent garbage collection revisited. In *OOPSLA*, pages 255–268, Nov. 2003.

[11] S.J. Baylor, M. Devarakonda, S.J. Fink, E. Gluzberg, M. Kalantar, P. Muttineni, E. Barsness, R. Arora, R. Dimpsey, and S.J. Munroe. Java server benchmarks. In *IBM Systems Journal*, 39(1):57–81, February 2000.

[12] BEA. BEA JRockit: Java for the enterprise. Technical white paper. http://www.bea.com, Jan. 2006.

[13] BEA Systems, Inc. *BEA Weblogic JRockit—The Server JVM: Increasing Server-side Performance and Manageability*, August 2002. `http://www.bea.com/products/weblogic/jrockit`.

[14] O. Ben-Yitzhak, I. Goft, E.K. Kolodner, K. Kuiper, and V. Leikehman. An algorithm for parallel incremental compaction. In *ISMM*, pages 207–212, Feb. 2003.

[15] S. Blackburn, P. Cheng, and K.S. McKinley. Myths and reality: The performance impact of garbage collection. In *SIGMETRICS*, pages 25–36, June 2004.

[16] S. Blackburn, P. Cheng, and K.S. McKinley. Oil and water? High performance garbage collection in Java with JMTk. In *ICSE*, pages 137–146, May 2004.

[17] S.M. Blackburn, R. Garner, C. Hoffmann, A.M. Khang, K.S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S.Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J.E.B. Moss, A. Phansalkar, D. Stefanovic, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA*, pages 169–190, Oct. 2006.

[18] S.M. Blackburn, M. Hertz, K.S. Mckinley, J.E.B. Moss, and T. Yang. Profile-based pretenuring. In *TOPLAS*, 29(1):2, 2007.

[19] S.M. Blackburn and A.L. Hosking. Barriers: Friend or foe? In *ISMM*, pages 143–151, Oct. 2004.

[20] S.M. Blackburn and K.S. McKinley. In or out?: Putting write barriers in their place. In *ISMM*, pages 281–290, June 2002.

[21] S.M. Blackburn and K.S. McKinley. Ulterior reference counting: Fast garbage collection without a long wait. In *OOPSLA*, pages 344–358, Oct. 2003.

[22] M.D. Bond and K.S. McKinley. Continuous path and edge profiling. In *MICRO*, pages 130–140, Dec. 2005.

[23] M.D. Bond and K.S. McKinley. Bell: Bit-encoding online memory leak detection. In *ASPLOS*, pages 61–72, Oct. 2006.

[24] M.D. Bond and K.S. McKinley. Probabilistic Calling Context in *OOPSLA*, pages 97–112, Oct. 2007.

[25] P. Bose. Performance evaluation and validation of microprocessors. In *SIGMETRICS*, pages 226–227, May 1999.

[26] K.R. Bowers and D. Kaeli. Characterizing the SPECjvm98 benchmarks on the Java Virtual Machine. Technical Report ECE-CEG-98-026, Northeastern University, Department of Electrical and Computer Engineering, September 1998.

[27] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A portable programming interface for performance evaluation on modern processors. In *The International Journal of High Performance Computing Applications*, 14(3):189–204, 2000.

[28] J.M. Bull, L.A. Smith, M.D. Westhead, D.S. Henty, and R.A. Davey. A benchmark suite for high performance Java. In *Concurrency, Practice and Experience*, 12(6), pages 375–388, May 2000.

[29] M.G. Burke, J-D. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M.J. Serrano, V.C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño dynamic optimizing compiler for Java. In *Proceedings of the ACM 1999 conference on Java Grande*, pages 129–141. ACM, 1999.

[30] D. Buytaert, A. Georges, M. Hind, M. Arnold, L. Eeckhout, and K. De Bosschere. Using HPM-sampling to drive dynamic compilation. In *OOPSLA*, pages 553–568, Oct. 2007.

[31] H.W. Cain, R. Rajwar, M. Marden, and M.H. Lipasti. An architectural evaluation of Java TPC-W. In *HPCA*. IEEE Computer Society, 2001.

[32] J. Cavazos and J. Moss. Inducing heuristics to decide whether to schedule. In *PLDI*, pages 183–194, June 2004.

[33] K. Chow, A. Wright, and K. Lai. Characterization of Java workloads by principal components analysis and indirect branches. In *WWC-1998, held in conjunction with MICRO-31*, pages 11–19, November 1998.

[34] M. Cierniak, M. Eng, N. Glew, B. Lewis, and J. Stichnoth. The open runtime platform: A flexible high-performance managed runtime environment. *Intel Technology Journal*, 7(1):5–18, 2003.

[35] W.G. Cochran. Sampling Techniques. John Wiley & Sons, 1977.

[36] R. Desikan, D. Burger, and S.W. Keckler. Measuring experimental error in microprocessor simulation. In *ISCA-28*, pages 266–277, July 2001.

[37] A.S. Dhodapkar and J.E. Smith. Managing multi-configuration hardware via dynamic working set analysis. In *ISCA*, pages 233–244. IEEE Computer Society, 2002.

[38] A.S. Dhodapkar and J.E. Smith. Comparing program phase detection techniques. In *ISCA*, page 217. IEEE Computer Society, 2003.

[39] K. Diefendorff. K7 challenges Intel. *Microprocessor Report*, 12(14), October 1998.

[40] M. Dmitriev. Selective profiling of Java applications using dynamic bytecode instrumentation. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*. IEEE Computer Society, 2004.

[41] E. Duesterwald, C. Cascaval, and S. Dwarkadas. Characterizing and predicting program behavior and its variability. In *PACT*, pages 220–231. IEEE Computer Society, 2003.

[42] B. Dufour, K. Driesen, L. Hendren, and C. Verbrugge. Dynamic metrics for Java. In *OOPSLA*, pages 149–168. ACM, 2003.

[43] L. Eeckhout, A. Georges, and K. De Bosschere. How Java programs interact with virtual machines at the microarchitectural level. In *OOPSLA*, pages 169–186, Oct. 2003.

[44] L. Eeckhout, H. Vandierendonck, and K. De Bosschere. Designing workloads for computer architecture research. *IEEE Computer*, 36(2):65–71, February 2003.

[45] L. Eeckhout, H. Vandierendonck, and K. De Bosschere. Quantifying the impact of input data sets on program behavior and its applications. *Journal of Instruction-Level Parallelism*, 5:1–33, February 2003. `http://www.jilp.org/vol5`.

[46] J. Fenn and A. Lindon. Gartners hype cycle report. http://www.gartner.com, 2004.

[47] S.J. Fink and F. Qian. Design, implementation and evaluation of adaptive recompilation with on-stack replacement. In *CGO* , pages 241–252. IEEE Computer Society, 2003.

[48] A. Georges, M. Christiaens, M. Ronsse, and K. De Bosschere JaRec: a portable record/replay environment for multi-threaded Java applications In *Software – Practice and Experience*, 34, pages 523–547, Feb. 2004

[49] A. Georges, D. Buytaert, and L. Eeckhout Method-Level Phase Behavior in Java Workloads. In *OOPSLA*, pages 270–287, Oct. 2004

[50] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous Java performance evaluation. In *OOPSLA*, pages 57–76, Oct. 2007.

[51] D. Gu, C. Verbrugge, and E.M. Gagnon. Relative factors in performance analysis of Java virtual machines. In *VEE*, pages 111–121, Jun. 2006.

[52] D. Gu and C. Verbrugge. Phase-based Adaptive Recompilation in a JVM. In *CGO*, pages , 2008

[53] S.Z. Guyer, K.S. McKinley, and D. Frampton. Free-me: A static analysis for automatic individual object reclamation. In *PLDI*, pages 364–375, Jun. 2006.

[54] M. Hauswirth, P.F. Sweeney, A. Diwan, and M. Hind. Vertical profiling: Understanding the behavior of object-priented applications. In *OOPSLA*, pages 251–269, Oct. 2004.

[55] M. Hauswirth, A. Diwan, P. Sweeney, and M.C. Mozer Automating Vertical Profiling In *OOPSLA*, pages 281–296, Oct. 2005.

[56] J.L. Hintze, and R.D. Nelson. Violin Plots: A Box Plot-Density Trace Synergism In *The American Statistician*, Volume 52(2), pages 181–184, May 1998.

[57] C.A. Hsieh, M.T. Conte, T.L. Johnson, J.C. Gyllenhaal, and W.W. Hwu. A study of the cache and branch performance issues with running Java on current hardware platforms. In *COMPCON*, pages 211–216, Feb. 1997.

[58] C.A. Hsieh, J.C. Gyllenhaal, and W.W. Hwu. Java bytecode to native code translation: The Caffeine prototype and preliminary results. In *MICRO*, pages 90–99, Dec. 1996.

[59] X. Huang, S.M. Blackburn, K.S. McKinley, J.E.B. Moss, Z. Wang, and P. Cheng. The garbage collection advantage: Improving program locality. In *OOPSLA*, pages 69–80, Oct. 2004.

[60] M.C. Huang, J. Renau, and J. Torrellas. Positional adaptation of processors: application to energy reduction. In *Proceedings of the 30th annual international symposium on Computer architecture*, pages 157–168, 2003.

[61] International Business Machines IBM Developer Kit and Runtime Environment, Java Technology Edition, Version 5.0 `http://download.boulder.ibm.com/ibmdl/pub/software/dw/jdk/diagnosis/diag50.pdf`

[62] R.A. Johnson and D.W. Wichern Applied Multivariate Statistical Analysis Prentice Hall, 2002

[63] The Kaffe virtual machine `http://www.kaffe.org`

[64] M. Karlsson, K.E. Moore, E. Hagersten, and D.A. Wood. Memory system behavior of Java-based middleware. In *HPCA*, pages 217–228, Feb. 2003.

[65] C. Krintz and B. Calder. Using annotations to reduce dynamic optimization time. In *PLDI*, pages 156–167, May 2001.

[66] J. Lau, S. Schoenmackers, and B. Calder. Structures for phase classification. In *ISPASS*, pages 57–67, 2004.

[67] L. Lamport Time, clocks, and the ordering of events in a distributed system. In *Communications of the ACM*, 21(7), pages 558–565, Jul. 1978.

[68] B. Lee, K. Resnick, M.D. Bond, and K.S. McKinley. Correcting the dynamic call graph using control-flow constraints. In *CC*, pages 80–95, Mar. 2007.

[69] T. Li, L.K. John, V. Narayanan, A. Sivasubramaniam, J. Sabarinathan, and A. Murthy. Using complete system simulation to characterize SPECjvm98 benchmarks. In *ICS*, pages 22–33, May 2000.

[70] T. Li, L.K. John, A. Sivasubramaniam, N. Vijaykrishnan, and J. Rubio. Understanding and improving operating system effects in control flow prediction. In *ASPLOS*, pages 68–80, Oct. 2002.

[71] D.J. Lilja. *Measuring Computer Performance: A Practitioner's Guide*. Cambridge University Press, 2000.

[72] Y. Luo, J. Rubio, L.K. John, P. Seshadri, and A. Mericas. Benchmarking internet servers on superscalar machines. In *IEEE Computer*, 36(2), pages 34–40, 2003.

[73] Y. Luo, and L.K. John. Efficiently Evaluating Speedup Using Sampled Processor Simulation. In *IEEE Computer Architecture Letters*, 3, 2004

[74] J. Maebe, M. Ronsse and K. De Bosschere. DIOTA: Dynamic Instrumentation, Optimization and Transformation of Applications. In *Compendium of Workshops and Tutorials Held in conjunction with PACT*, Sep. 2002.

[75] J. Maebe, D. Buytaert, L. Eeckhout, and K. De Bosschere. Javana: A system for building customized Java program analysis tools. In *OOPSLA*, pages 153–168, Oct. 2006.

[76] B.F.J. Manly. *Multivariate Statistical Methods: A primer*. Chapman & Hall, second edition, 1994.

[77] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine. In *Communications of the ACM*, 3, pages 184–195, 1960.

[78] P. McGachey and A.L. Hosking. Reducing generational copy reserve overhead with fallback compaction. In *ISMM*, pages 17–28, Jun. 2006.

[79] McQuitty, L.L. Similarity Analysis by Reciprocal Pairs for Discrete and Continuous Data. In *Educational and Psychological Measurement*, 26, pages 825–831, 1966.

[80] M.C. Merten, A.R. Trick, R.D. Barnes, E.M. Nystrom, Christopher N. George, John C. Gyllenhaal, and Wen mei W. Hwu. An architectural framework for run-time optimization. *IEEE Transactions on Computers*, 50(6), pages 567–589, 2001.

[81] P. Nagpurkar, C. Krintz, M. Hind, P.F. Sweeney, and V.T. Rajan. Online phase detection algorithms. In *CGO*, pages 111–123, 2006

[82] J. Neter, M.H. Kutner, W. Wasserman, and C.J. Nachtsheim Applied Linear Statistical Models WCB/McGraw-Hill, 1996.

[83] K. Ogata, T. Onodera, K. Kawachiya, H. Komatsu, and T. Nakatani. Replay compilation: Improving debuggability of a just-in-time compiler. In *OOPSLA*, pages 241–252, Oct. 2006.

[84] M. Paleczny, C. Vick, and C. Click. The Java Hotspot server compiler. In *JVM*, pages 1–12, Apr. 2001.

[85] D. Pelleg, A. Moore X-means: Extending K-means with Efficient Estimation of the Number of Clusters In *ICML*, pages 727–734, 2000.

[86] D. G. Perez, G. Mouchard, and O. Temam MicroLib: A Case for the Quantitative Comparison of Micro-Architecture Mechanisms In *MICRO*, pages 43–54, 2004.

[87] R. Radhakrishnan, N. Vijaykrishnan, L.K. John, and A. Sivasubramaniam. Architectural issues in Java runtime systems. In *HPCA*, pages 387–398, January 2000.

[88] R. Radhakrishnan, N. Vijaykrishnan, L.K. John, A. Sivasubramaniam, J. Rubio, and J. Sabarinathan. Java runtime systems: Characterization and architectural implications. *IEEE Transactions on Computers*, 50(2):131–145, February 2001.

[89] R Development Core Team. *R: A language and environment for statistical computing*. R Foundation for Statistical Computing, Vienna, Austria, 2003. ISBN 3-900051-00-3.

[90] M. Rosenblum, E. Bugnion, S. Devine, and S.A. Herrod. Using the SimOS machine simulator to study complex computer systems. In *TOMACS*, 7(1):78–103, January 1997.

[91] E. Gagnon and L. Hendren SableVM: A Research Framework for the Efficient Execution of Java Bytecode. In *JVM01*, pages 27–40, 2001.

[92] N. Sachindran and J.E.B. Moss. Mark-copy: Fast copying GC with less space overhead. In *OOPSLA*, pages 326–343, Oct. 2003.

[93] N. Sachindran, J.E.B. Moss, and E.D. Berger. MC2: high-performance garbage collection for memory-constrained environments. In *OOPSLA*, pages 81–98, Oct. 2004.

[94] K. Sagonas and J. Wilhelmsson. Mark and split. In *ISMM*, pages 29–39, June 2006.

[95] N. Schenker, J.F. Gentleman. On Judging the Significance of Differences by Examining the Overlap Between Confidence Intervals. In *The American Statistician*, Vol. 55 (3), pages 182–186, 2001.

[96] F.T. Schneider, M. Payer, and T.R. Gross. Online optimizations driven by hardware performance monitoring. In *PLDI*, pages 373–382, June 2007.

[97] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *ASPLOS*, pages 45–57. ACM, 2002.

[98] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. In *ISCA*, pages 336–349. ACM, 2003.

[99] Y. Shuf, M.J. Serrano, M. Gupta, and J.P. Singh. Characterizing the memory behavior of Java workloads: a structured view and opportunities for optimizations. In *SIGMETRICS*, pages 194–205. Jun. 2001.

[100] D. Siegwart and M. Hirzel. Improving locality with parallel hierarchical copying GC. In *ISMM*, pages 52–63, June 2006.

[101] S. Soman, C. Krintz, and D.F. Bacon. Dynamic selection of application-specific garbage collectors. In *ISMM*, pages 49–60, June 2004.

[102] Standard Performance Evaluation Corporation. SPECjbb2000 benchmark http://www.spec.org/jbb2000.

[103] Standard Performance Evaluation Corporation. SPECjvm98 Benchmarks. http://www.spec.org/jvm98.

[104] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani. Overview of the IBM Java Just-in-Time compiler. In *IBM Systems Journal*, 39(1):175–193, February 2000.

[105] T. Suganuma, T. Yasue, M. Kawahito, H. Komatsu, and T. Nakatani. Design and evaluation of dynamic optimizations for a Java just-in-time compiler. In *TOPLAS*, 27(4):732–785, Jul. 2005.

[106] V. Sundaresan, D. Maier, P. Ramarao, and M. Stoodley. Experiences with multithreading and dynamic class loading in a Java just-in-time compiler. In *CGO*, pages 87–97, Mar. 2006.

[107] Sun Microsystems, Inc. *The Java HotSpot Virtual Machine, v1.4.1*, September 2002.

[108] P.F. Sweeney, M. Hauswirth, B. Cahoon, P. Cheng, A. Diwan, D. Grove, and M. Hind. Using hardware performance monitors to understand the behavior of Java applications. In *VM*, pages 57–72, May 2004.

[109] J. Whaley. A portable sampling-based profiler for Java virtual machines. In *Proceedings of the ACM 2000 Conference on Java Grande*, pages 78–87, June 2000.

[110] T. Yang, M. Hertz, E.D. Berger, S.F. Kaplan, and J.E.B. Moss. Automatic heap sizing: taking real memory into account. In *ISMM*, pages 61–72, June 2004.

[111] C. Zhang, K. Kelsey, X. Shen, C. Ding, M. Hertz, and M. Ogihara. Program-level adaptive memory management. In *ISMM*, pages 174–183, June 2006.